

C++程序设计系列教材

C++程序设计教程

（第3版）通用版

◎ 钱能 著

电子课件

程序源码



清华大学出版社

C++程序设计系列教材

C++程序设计教程(第3版)
通用版

钱能　著

清华大学出版社
北京

内 容 简 介

C++是一种高效实用的程序设计语言，它既可进行过程化程序设计，也可进行面向对象程序设计，因而成为编程人员最广泛使用的工具。学好 C++，再学习其他软件就很容易，C++架起了通向强大、易用、真正的软件开发应用的桥梁。本书共分两大部分：第一部分包括第 1 章～第 10 章，是基础部分，主要介绍 C++程序设计语言、程序结构和过程化基础；第二部分包括第 11 章～第 21 章，是面向对象程序设计部分，它建立在 C++程序设计基础之上，讲述了面向对象程序设计方法。

本书提供课程教学的全程视频，读者可扫描封底的刮刮卡观看。本书还提供电子课件和程序源码，读者可以扫描封底的课件二维码下载。

本书适合用作大学计算机专业和非计算机专业的程序设计基础课程教材，也可供自学的读者使用。

图书在版编目(CIP)数据

C++程序设计教程：通用版/钱能著. —3 版. —北京：清华大学出版社，2019
(C++程序设计系列教材)
ISBN 978-7-302-52126-6

Ⅰ. ①C…　Ⅱ. ①钱…　Ⅲ. ①C 语言－程序设计－高等学校－教材　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2019)第 009721 号

策划编辑：魏江江
责任编辑：王冰飞
封面设计：刘　键
责任校对：徐俊伟
责任印制：李红英
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市龙大印装有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：29.75　　**字　　数**：721 千字
版　　次：1999 年 4 月第 1 版　2019 年 6 月第 3 版　　**印　　次**：2019 年 6 月第1 次印刷
印　　数：1～3000
定　　价：69.80 元

产品编号：081890-01

第3版前言

人工智能发展的大势

人工智能代表了人类科学发展的前沿领地，C++与其关系密不可分，所以本教材系列的出版有人工智能发展背景的一席之地。

人工智能目前尚处初级阶段，但其研究所派生的应用已经硕果累累，正在快速地改变我们的生活。人工智能解读医学拍片的本领已经比医生高；查阅法律证据的能力也比律师强；飞机及航空管理正在被人工智能替代；车辆行驶人工智能系统比人的操纵更好；搜索引擎中的人工智能可以分析照片，告诉你照片里面的故事。在线地图、数码相机、自动驾驶、无人超市、无人餐馆、无人银行等，今后甚至桩桩、件件、处处都可装智能芯片，从而纳入人工智能管理。

人工智能最关心的是人工自主意识，目前网络和计算机已经完成了知识的检索和存储，几大搜索引擎也完成了关键字-关联解释的功能和海量数据积累，大多数机器人厂商已经完成了反应机、自适应等高级功能，但却还没有能通过图灵测试的真正的人工自主意识。当然人类对自身意识的研究水平制约着人工智能的实现，人工智能的应用还可反哺于人类对自身意识的研究。

人工智能或许认为，神经网络系统只有复杂到一定程度，且在大尺度上的相似性保持高度一致，其个体自然产生的意识才会具备类似神经网络个体的认同和感知。但在技术上，意识只不过是人造神经网络中诸多需求反馈链交错所致。所以，人们通过研究人类神经网络的构成分布、互联网的社会化训练过程，"自然产生"个体意识。但实际上目前网上的软件自动机和各种设备产生的不知名网络现象，即所谓自主意识，因还无法被人工智能所感知，只被当作不知名故障进行"修复"处理，自当无解。

人工智能又或许认为，可以通过人工制造的智慧个体（机器人），在初期表现出类似创造者的行为和意识，再慢慢地进化。"机器学习"和"深度学习"被证明是个有效的手段，但受限于机器人硬件发展和大数据，前路漫漫。况且面临着神经反馈网络发展的实际问题，进化过程中的数据"过载"或"饥荒"会导致行为和意识的随时失却。

然而人类正在不依不饶地解决人工智能的关键问题：机器人的行动能力和对环境的视觉、听觉、触觉、嗅觉感知能力都在快速增强，智能推演之，则机器人就可自行获取运行的能源；软件自编程系统逐渐实现的自继承、自升级和自恢复，可以使机器人自我修复和完善；人类所掌握的全方位机器人设计、生产、测控在逐渐人工智能化，总有一天，机器人可以自行复制。

未来的人工智能发展速度将呈指数级攀升，将有越来越多的机器人通过图灵测试而具意识。一旦人工智能具有创造性思维，其发展将促进人类的巨大科学进步。显然，人工智能



离不开计算,其需要远远大于现有的计算能力,除了期待量子计算机外,还需要有高可靠性的软件架构和高性能算法,这便需要千锤百炼的编程语言和纵横交错的软件工具。

C++发展与地位

C++在1998年制定了一个里程碑式的C++98国际标准,确立了C++语言的强势地位。之后,C++标准每年修订,2011年制定的C++11标准,使得C++的强类型特征得到了充分的体现,模板编程规范渐趋成熟。C++11标准再次深度影响了C++编译器的变革,其发展无时无刻不在说明其语言的完美缔造。

C++充分继承了C,保持了与硬件的亲和性,在此基础上,有机结合了诸多编程方法,兼容C的过程化编程框架,实现了面向对象的高效设计,又开辟了可自动生成的模板编程架构,在程序设计语言界绝无仅有。C++是当前使用最广泛的软件工具之一,其实现技术含量最高,应用于最重要领域。C++给我们搭建的软件架构,得以让人类展开多层次的人-人、人-机的互动设计,其正完美地表现出作为人类自然语言的化身角色。

从另一个角度来说,C++编程本身就是在撰写一篇优美的诗文,叙述一个精彩的故事,谱写一首动听的曲子。随着韵律和情节的跌宕起伏,什么时候故事讲完了,代码也就收尾了。好文章语义清晰、简练生动、词藻华美、引人入胜;好代码通俗易懂、结构清晰、层次分明、优化高效。因为C++独具多种编程方法,包揽从算法优化的微观细节,到模板架构的宏观布局,因而其开拓了编程中更广泛的遐想与表达的空间,C++编程充满美感。

微软操作系统及其架构,Apple的大部分底层软件,腾讯的QQ和微信,阿里云、百度云计算之底层架构,Google的Android底层架构,大部分数据库核心代码,几乎所有重要的系统,只要上规模,需要保证高可靠性,计较性能,无一不是用C++工具搭建。

正因为C++继承了C的衣钵,充分实现与系统硬件的无缝对接,追求高效率编程,才使得人工智能兴起的今天,大量涉及硬件相关的软件开发,C++是首选;其在人工智能的软件架构中,核心的逻辑语义表达,不但描述能力无可挑剔,而且在性能和效率方面占尽了优势。

重量级IT企业在招聘大数据工程师时,机器人公司在招聘开发人员时,都把C++编程作为必备能力。目前在中小学教学的信息学与程序设计课程开设中,C++趋向于统一指定为高考入学备考科目。事实上,学好C++,再自学其他编程语言就很容易,反之则不行。

编程语言的世界排名前四名已经长时间被Java、C、C++、Python这4种语言所占据。Java因其应用面更广泛而持续居于榜首,但在人工智能领域,Python编程相比Java,或许更加清爽、整洁、漂亮,其跃居前四,又有后来居上之势。人工智能也带来了C++的再次繁荣,从某种程度上说,Python编程只是在搭建软件的外包装,而C++才是其核心。C++与C在占据系统底层应用方面没有什么差距,但是在规模化编程、自动生成、实现系统架构方面,非C++莫属。况且由于C++源自C的特点,C编程往往又是在C++平台中实现。追本溯源,C++语言才是当今人工智能大发展上最重要的工具。

改版框架

本教材系列进化到第3版,是作者20多年C++教学研究与实践的总结。改版之后,每



本主教材的框架结构没有变，所以遵循原编排特点、内容特点、学习方式。但毕竟编程应用需求形势大变，C++的地位攀升，急需权威的 C++教材主导 C++的编程教学，故而第 3 版各版本的名称拟定、排版、内容都作了较大更新。

第 3 版中各版本一律改用双色文字排版，代码以及关注文字用另一种颜色和底纹凸显，从根本上改变了排版式样，可读性得以显著提升。

第 3 版中各版本的内容在原书的基础上修改提升，涉及内涵深度、风格表现、描述侧重点等诸多不同。其版本名称见表 1。

表 1　第 3 版版本框架

序	类别	较　早　版	新　　版
1	基础型主教材	《C++程序设计教程(修订版)——设计思想与实现》(十二五规划教材)	《C++程序设计教程(第 3 版)通用版》
2	实战型主教材	《C++程序设计教程(第 2 版)》(十一五规划教材)	《C++程序设计教程(第 3 版)竞技版》
3-1	拓展型主教材	《C++程序设计教程详解——过程化编程》(十一五规划教材)	《C++程序设计教程(第 3 版)专业版——过程化编程》
3-2	拓展型主教材	《C++程序设计教程详解——对象化编程》* (十一五规划教材)	《C++程序设计教程(第 3 版)专业版——对象化编程》
4	配套教辅	《C++程序设计教程(第 2 版)——实验指导》(十一五规划教材)	《C++程序设计教程(第 3 版)——实验指导》
5	配套教辅	《C++程序设计教程(第 2 版)——习题及解答》(十一五规划教材)	《C++程序设计教程(第 3 版)——习题解答》

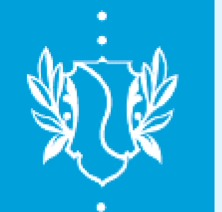
* 指原书未出版。

第 3 版的通用版：侧重 C++基础，主要从概念着手，介绍 C++编写程序的技法，强调编写正确的程序。学习之后，应当能了解 C++是怎么回事，能解决什么问题，能看懂 C++程序，了解 C++的诸多技术特征，能编制一些简单的 C++程序，能发现一些常规的 C++错误，了解不同的程序设计方法，对面向对象程序设计方法及其特征有一个基本的了解，具备进一步学习后续课程(如数据结构、算法分析与设计)的基础。

第 3 版的竞技版：侧重 C++分析设计技术，从实战训练着手，介绍 C++的各种编程策略与技术，引导对数学及算法学习的重视，强调编写高效的程序。学习之后，应当能掌握基本的问题分析方法，掌握解决问题的设计技术；了解编程过程中的许多难点，深切体会细节决定成败；能够学习且具备参加各个层次程序设计竞赛的能力；对 C++能解决什么问题的能力有全新的看法，进一步了解面向对象程序设计的方法；学会层次分析和功能拆解，具备独立设计一个规模较大的程序的能力；具备语言学习的独立能力。

第 3 版的专业版：一方面对竞技版的 C++分析设计技术从底层的内存布局、编译器类型识别、各项技术相互关联等进行深度解析；另一方面介绍 C++新标准及其新编译器所涉及的技术，以纵向视角来审视 C++的未来发展，更全面地了解 C++的实现技术，全面了解面向对象程序设计方法和技术，产生对高级模板编程的兴趣。虽然本版本未必能成为高校 C++课程学习的主流，但是将其作为参考，可以作为国外诸多 C++优秀教材之补充。

通用版、竞技版、专业版编纂目的不同，学习目标不同，但 3 个版本都出自同一起点——



“Hello World”。每个主教材版本独立成体系,保证概念的正确性和前后连贯性,而又相互补充,展示 C++不同的发展阶段,也展示不同的目标要求,满足了不同学习能力的读者的学习需要。对于没有编程基础的读者,则适合从基础型教材的学习开始,逐渐进入实战型教材的学习训练,而将拓展型教材作为研读或参考教材,去领略 C++前沿之精妙。

在上述 3 个版本主教材的基础上,所撰写的三大教材的统一的实验指导和习题解答,则适合作教辅资料。倘若没有基础版的学习,又无行家点拨,则后面的编程学习会具有一定的困难,这也是在教学过程中确实存在的问题。

第 3 版的教材与其他国内外教材最大的不同,是聚焦于培养读者的编程实战能力。C++语法现象的学习或许并没有面面俱到,但是运用 C++的编程方法与技巧,实际地解决问题,却占有相当的篇幅。

本书技术特征

本书保持了原书的描述风格,知识密集型特征,将基本概念有效地划分为一个个独立块,形式多样地讲清,而形成结构清晰的亮点,独到地按从简单到复杂、从易到难循序渐进地推进各个章节。

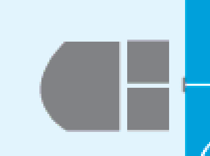
成书较早而使用 BC、BCB 6.0、VC 6.0 这 3 种编译器。代码风格常有 16 位编译器的影子。例如,浮点变量采用 float 居多,未涉 64 位整型等。因反映的是 C++98 标准前后的内容,故与 C 结合得紧密些。改版后,代码重写而逐渐形成自己的风格,且全部符合 C++98 标准。然而,C 语言描述习惯的痕迹仍在。例如,变量定义统统放在块首,for 循环变量仍早在 for 之前定义。

改版坚持把概念准确放在第一位。书中调整了个别章节,例如,在第 11 章添加了“名字识别”这一节,将原第 17 章的“多重继承”并到了第 16 章,而原第 16 章的“多态”和“抽象类”独立出来成为第 17 章。特别在面向对象程序设计部分,修改和明确了多处原概念模糊不清的地方。例如,模板类与类模板的区别,以及与类模板实例的差别,私有继承与保护继承的概念,多态的目的,继承与组合的差别与联系,等等。调整之后,理顺知识、归类概念变得更加自然。

改版继承了原书诸多优点。例如,在实例应用方面,强调完整实现,除了连贯地使用了单一问题描述的系列解决方案 Josephus 之外,还对诸多排序方法以及链表算法作了详细的描述;关于字串处理,从字符的数组、指针、库函数,到输入/输出,甚至字串流,代码全部使用 C 字串,系统滤清了 C 字串的概念;书中重视操作符重载,用了许多实例,滤清了多种单目操作符和双目操作符的用法和使用误区;书中对“引用”概念单辟一章来写,专述其原理及使用;此外,注重描述一些重要概念,如左值、类型相容及显隐式转换、表达式副作用等,这些都是编程最容易误解或出错之处,构成 C++进阶的重要基础。

因通用版关乎 C++基础,故未涉及 C++STL 概念,也未述及 string 字串。其编程尚停留在程序正确性之品质要求。面向对象程序设计概念也需要后续版本进一步展开。例如,在多态编程中,仅简单叙述多态对象类型转换的概念。

教材注重能力培养的理念与架构,必然在课程教学中从事问题驱动的教学模式,重视实践环节的设计和辅导,故在前言后面的附表中列出了课程教学的全程视频对应表,读者可扫



描封底的刮刮卡观看教学视频。本书还提供电子课件和程序源码，读者可以扫描封底的课件二维码下载。

温馨致谢

世界真奇妙，人逢知天命之年，却还百般任性，人的劣性也由此爆发，各种不顺都来围剿，整天疲于应对琐事，因而我放弃了写书。不料，出版社的魏江江，一句希望，一句怂恿，把我封存在心的C++情结给钓了出来。回想这改版啊，本来就是我的梦。终于2018年的9月，决定要做改版的事。

编辑王冰飞，诸多鼓励和建议，洋溢着热情与幽默，以及印象深刻的高效工作，让我感受到C++教材撰写工作的崇高。教材的受益群体，从中小学生、大学生到程序员，都需要提升编程能力来强化自己的内涵和跟进现实世界，以致我认定了意义，直奔赶进度的节奏而去。

家人默默的生活支持，酿成了一种影响力，一句“快写C++”的催促，将本不起眼的几个音节，窝成了一个大大的推波，汹涌地扑在我的心上。

诸多同事，C++的OPS(Online Programming Space在线编程天地)提交系统维护者刘端阳和陈波老师，还有与我抱团的张永良和王英姿老师，共同实施了C++精品课程的编程能力训练。还有我的整个C++教学团队，学院教学院长江颉老师从一开始就是C++能力培养型教学的支持者，课程教学责任人龙胜春老师的虚心求教的启示及平时给予我很多的关照，毛国红老师对我的C++程序设计试卷提出的精益求精的见解，等等，恕不一一列出。他们都是我教材撰写的促进者。

在我内心深处，还有一种更原始的动力，来自恩师王国东先生，他是我的人生导师，我能得以轻轻放下，又重拾信心，拣起改版一事；他更是我长此以往写C++教材时的诸多灵感与智慧的源泉，系列成书，功不可没第一人。

钱　能

于杭州自在居

2019年除夕

附表 1　C++视频对应表——过程化编程

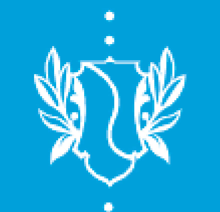
序	标题	注　　释
1-001	C++课程概述	C++课程的编程概述,实验方法
1-002	编程操作提交	编程操作,提交平台,实验 1 布置
1-003	输入输出和循环	简单语句,变量与字符,循环
1-004	变量与字符	字符,字符三角形
1-005	次数控制循环	编程细节,字符菱形 1
1-006	增量操作	增量操作,字符菱形 2
1-007	输出格式	交替字符倒三角形,格式阵列 1
1-008	整型原理	格式阵列 2,整型
1-009	1! 到 n! 的和	1! 到 n! 的和 1
1-010	文件操作	1! 到 n! 的和 2,最大公约数,文件操作
1-011	浮点输出	浮点格式输出,等比数列 1,斐波那契数列
1-012	函数使用	表达式副作用,函数,最大公约数,最小公倍数,寻找素数对 1
1-013	素数筛法	寻找素数对 2,素数筛法,对称三位数素数 1,逻辑短路
1-014	浮点型原理 1	浮点型 1
1-015	浮点型原理 2	浮点型 2,级数求和
1-016	集合	逻辑短路,集合,对称三位数素数 2
1-017	位操作	对称三位数素数 3,位操作,整数内码,整除 3、5、7
1-018	递归 1	整除 3、5、7,母牛问题,递归
1-019	空间换时间	A 类数,协方差 1
1-020	数学方法优化	协方差 2,五位以内对称素数
1-021	提交策略	做题提交策略,十-二进制转换 1
1-022	转移语句	转移语句,十-二进制转换 2,统计天数
1-023	字串处理	字符,字串处理,输出格式
1-024	计算技巧	uglyNumber
1-025	期中讲评 1	期中考试讲评,接龙,斜纹布,斐波追溯数 1
1-026	期中讲评 2	斐波追溯数 2,字符表,少数服从多数
1-027	期中讲评 3	11 的倍数,无秤售油,组合数 1
1-028	期中讲评 4	组合数 2,矩阵鞍点 1
1-029	多重集	列出完数,12! 配对 1
1-030	二维数组	12! 配对 2,矩阵鞍点 2
1-031	排序 1	排序 1,参数传递
1-032	排序 2	排序 2
1-033	结构	0-1 串排序,按绩点排名 1
1-034	逆反	按绩点排名 2,逆反 0-1 串
1-035	String 搜索 1	去掉双斜杠注释,string 串的 find,排列对称串
1-036	常规做题策略	BoxofBricks,算菜价
1-037	数学方法运用	n! 的位数
1-038	String 搜索 2	剪花布条
1-039	递归 2	勘探油田 1
1-040	Map	勘探油田 2,最多的商品
1-041	运行错误解析	Getline,运行错误解析

附表 2　C++视频对应表——面向对象编程

序	标题	注　　释
2-005	期末讲评与程序结构 1	C++程序设计Ⅰ期末考试讲解，过程化程序结构 1
2-006	程序结构 2	过程化程序结构 2
2-007	四则运算程序控制	PartⅡ第四套实验讲解——简单四则运算
2-008	大数加，计算器实验 1	大数加等，计算器样本实验问题理解，实验要求 1
2-009	n！中的 0	PartII 第五套实验讲解——n！中的 0
2-010	计算器实验 2	计算器样本实验处理总体框架逻辑
2-011	程序结构 3	过程化程序结构 3
2-012	程序结构 4	过程化程序结构 4，实验要求 2
2-013	计算器实验 3	计算器样本实验数据处理过程，实验要求 3
2-014	计算器实验 4	计算器样本实验程序控制，测试数据制作
2-015	类与对象 1	数据类型，数据传递，数据封装 1
2-016	类与对象 2	数据封装 2，对象创建，计算器实验-类型创建
2-017	类与程序结构 1	对象化编程 1，计算器实验程序框架 1
2-018	类与程序结构 2	对象化编程 2，计算器实验程序框架 2，异常处理
2-019	对象创建 1	对象内存映射 1
2-020	对象创建 2	对象内存映射 2
2-021	对象创建 3	对象内存映射 3，深拷贝浅拷贝 1，拷贝构造 1，赋值
2-022	对象创建 4	动态内存申请，深拷贝浅拷贝 2，拷贝构造 2，析构
2-023	继承 1	访问权限，对象内存映射 4
2-024	继承 2	对象内存映射 5，批量数据处理特征 1
2-025	多态 1	批量数据处理特征 2，多态-虚函数 1
2-026	多态 2	函数重载与覆盖，多态-虚函数 2
2-027	多态处理	批量数据处理特征 3
2-029	抽象类 1	纯虚函数
2-030	抽象类 2	面向对象程序结构
2-031	归纳面向对象 1	学术竞赛讲评 1
3-001	归纳面向对象 2	学术竞赛讲评 2

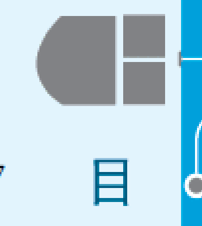
目录

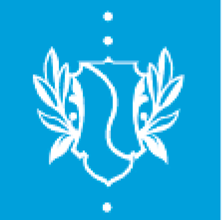
第一部分　C++过程化语言基础





第二部分　面向对象程序设计





第一部分

C++过程化语言基础



第1章　C++入门

C++是一门优秀的程序设计语言。C++比 C 更容易被人们所学习和掌握，并且以其独特的语言机制在计算机科学领域中得到广泛的应用。学习本章后，要求了解 C++语言的概念，了解 C 与 C++之间的关系，了解 C++语言对程序设计方法的支持，了解 C++程序开发的过程，了解简单的 C++程序结构，学会最简单的 C++程序开发。

1.1　从 C 到 C++

C 语言是贝尔实验室的 Dennis Ritchie 在 B 语言的基础上开发出来的，1972 年在一台 DEC PDP-11 计算机上实现了最初的 C 语言。C 是作为 UNIX 操作系统的开发语言而广为人们所认识的。实际上，当今许多新的、重要的操作系统都是用 C 或 C++编写的。C 语言是与硬件无关的。由于 C 语言的严谨设计，使得把用 C 语言编写的程序移植到大多数计算机上成为可能。到 20 世纪 70 年代末，C 已经演化为“传统的 C 语言”。Kernighan 和 Ritchie 在 1978 年出版的 *The C Programming Language* 一书中全面地介绍了传统的 C 语言，这本书已经成为最成功的计算机学术著作之一。

C 语言在各种计算机上的快速推广导致了许多 C 语言版本的出现。这些版本虽然是类似的，但通常是不兼容的。对希望开发出的代码能够在多种平台上运行的程序开发者来说，这是一个大麻烦。显然，人们需要一种标准的 C 语言版本。为了明确地定义与机器无关的 C 语言，1989 年美国国家标准协会制定了 C 语言的标准——ANSI C。Kernighan 和 Ritchie 编著的第二版 *The C Programming Language*（1988 年版）介绍了 ANSI C 的全部内容。

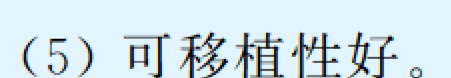
至此，C 语言以其独有的特点风靡了全世界：

（1）语言简洁、紧凑，使用方便、灵活。C 语言只有 32 个关键字，程序书写形式自由。

（2）丰富的运算符和数据类型。

（3）C 语言可以直接访问内存地址，能进行位操作，使其能够胜任开发操作系统的工作。

（4）生成的目标代码质量高，程序运行效率高。



(5) 可移植性好。

C 语言盛行的同时，也暴露出它的局限性：

(1) C 类型检查机制相对较弱，这使得程序中的一些错误不能在编译时发现。

(2) C 本身几乎没有支持代码重用的语言机制，因此一个程序员精心设计的程序，很难为其他程序所用。

(3) 当程序的规模达到一定的程度时，程序员很难控制程序的复杂性。

为了满足管理程序的复杂性需要，1980 年，贝尔实验室的 Bjarne Stroustrup 开始对 C 进行改进和扩充。最初的成果称为“带类的 C”，1983 年正式取名为 C++，在经历了 3 次 C++修订后，于 1994 年制定了 ANSI C++标准的草案。以后又经过不断完善，成为目前的 C++。C++仍在不断发展中。

C++包含了整个 C，C 是建立 C++的基础。C++包括 C 的全部特征、属性和优点，同时添加了对面向对象编程(OOP)的完全支持。

1.2 程序与语言

1. 程序

程序是以某种语言为工具编制出来的动作序列，它表达了人的思想。计算机程序是用计算机程序设计语言所要求的规范书写出来的一系列动作，它表达了程序员要求计算机执行的操作。

对于计算机来说，一组机器指令就是程序。当我们说机器代码或者机器指令时，都是指的程序，它是按计算机硬件设计规范的要求编制出来的动作序列。

对于使用计算机的人来说，程序员用某高级语言编写的语句序列也是程序。程序通常以文件的形式保存起来，所以，源文件、源程序和源代码都是程序。

程序是任何有目的的、预想好的动作序列。它构成软件。

计算机要运转起来，需要一整套可运行软件，即计算机程序。

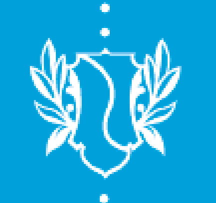
学术界对程序的定义是比较严格的，这里不作详述。

2. 程序语言的发展

最早，程序员使用最原始的计算机指令，即机器语言程序。只有机器语言才能为机器所识别和运行。这些指令由一串二进制的数表示。不久，发明了汇编语言，它可以将机器指令映射为一些能被人读懂的助记符，如 ADD，SUB。程序员运行汇编程序将用助记符写成的源程序转换成机器指令，然后再运行机器指令程序，得到所要的结果。那时，编写程序的都是计算机专业人员，编写程序的语言都是低级的或较低级的。

以后，随着硬件的发展，Fortran、BASIC、Pascal、C 等几十种甚至几百种高级语言应运而生，中间经历了严酷的优胜劣汰过程，最后剩下的是一些比较优秀的高级语言。C++首当其冲。

多年来，计算机程序的主要目标是力求编写出短小的代码以使运行速度更快。因为硬件成本和上机运行费很高。当计算机变得更小、更廉价、运行速度更快时，计算机硬件和运



行的成本快速下降,而程序员开发程序、维护程序的费用却急剧上升,程序设计的目标也就发生了变化。

在程序正确的前提下,可读性、易维护、可移植是程序设计首要的目标。所谓可读,就是使用良好的书写风格和易懂的语句编写程序。所谓易维护,是指当业务需求发生变化时,不需要太多的开销就可以扩展和增强程序的功能。所谓可移植,是指编写的程序在各种计算机和操作系统上都能运行,并且运行结果一样。

3. 高级语言和低级语言

C++语言是高级语言,机器语言是低级语言,汇编语言基本上是低级语言。例如,对于C++语言的语句:

```
a = 3 * a - 2 * b + 1;                    //3a - 2b + 1 的值赋给 a
```

写成汇编语言和对应的机器语言为:

```
mov eax, DWORD PTR a_ $ [ebp]            8b 45 fc
lea eax, DWORD PTR [eax + eax * 2]       8d 04 40
mov ecx, DWORD PTR b_ $ [ebp]            8b 4d f8
add ecx, ecx                             03 c9
sub eax, ecx                             2b c1
inc eax                                  40
mov DWORD PTR a_ $ [ebp], eax            89 45 fc
```

第一条命令是将 a 放入寄存器 eax 中(ebp 是数据段的指针,a_ $ 是变量 a 的偏移位置)。

第二条命令是将 eax 的内容加上 2 倍的 eax 内容放到 eax 中,即 eax 中值为 3 * a。

第三条命令是将 b 放入寄存器 ecx 中。

第四条命令是将 ecx 的内容加上 ecx,即 ecx 中的值为 2 * b。

第五条命令是将 eax 减去 ecx 的值(3 * a－2 * b)放入 eax。

第六条命令是 eax 的值加 1。此时,eax 中的值为 3 * a－2 * b＋1。

最后一条命令是将寄存器 eax 的值放入 a 变量中,即实现 a＝3 * a－2 * b＋1。

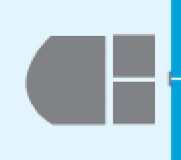
可以看出,程序语言越低级,描写程序越复杂,指令越难懂。语言越低级,就越靠近机器;语言越高级,就越靠近人的表达与理解。

程序语言的发展,总是从低级到高级,直到可以用人的自然语言来描述。

程序语言的发展,也是从具体到抽象的发展过程。编制一个表达式,无须将表达式的具体操作过程描述出来,否则人会感到太累,大量的精力会被无谓地浪费,无法进行更大规模的设计与思考;而低级语言则必须详尽地描述任何操作。所以,抽象表达能力越强,语言越高级。

4. C 与 C++

C++语言包括过程性语言部分和类部分。过程性语言部分与 C 并无本质的差别,无非版本提高了,功能增强了。类部分是 C 中所没有的,它是面向对象程序设计的主体。要学习面向对象程序设计,首先必须具有过程性语言的基础。所以**学习 C++,必先学习其过程性语言部分**,然后再学类部分。也就是说,先学高版本的 C,再学类。从过程性语言的共同具



有这个意义上来说，学习 C++，无须先学 C。

过程化程序设计基于结构化程序设计理论。在特定的 C++ 语境中，则必须依赖其函数框架、循环控制、分支结构与数据说明等描述。在本质上，过程化程序设计与结构化程序设计是一致的，只是前者比较率性，直接把编码当作设计；后者更规范，强调分析设计应走流程。而在方法上，结构化或过程化程序设计则作为独立的方法，区别于面向对象程序设计和基于模板程序设计。

从语言的能耐上来说，C 能很好地支持结构化程序设计，而 C++ 既能很好地支持结构化程序设计，又能很好地支持面向对象程序设计甚至模板化程序设计。所以，正如 C++ 的泰斗 Bjarne Stroustrup 所说："先学 C 没有必要。"

然而，C 语言程序设计的经验非常有益。因为 C 程序设计开发锻炼了程序员进行抽象程序设计的能力，这正是 C++ 更为抽象的概念和技术的基础。而且，C++ 是 C 语言的扩展，它分享了 C 的许多技术风格。C 程序设计特性在 C++ 中得到频繁使用。一个人使用 C 的经验越丰富，编写 C++ 程序也就越容易。所以，学过 C 能够促进 C++ 的学习。

1.3 结构化程序设计

以前，人们把程序看成是处理数据的一系列过程。过程或函数定义为一个接一个顺序执行的一组指令。数据与程序分开存储，编程的主要技巧在于追踪哪些函数调用哪些函数，哪些数据发生了变化。为解决其中可能存在的问题，结构化编程应运而生。

结构化程序设计的主要思想是功能分解并逐步求精。当一些任务十分复杂以致无法描述时，可以将它拆分为一系列较小的功能部件，直到这些自完备的子任务小到易于理解的程度。例如，计算一个公司中每一个职员的平均工资是一项较为复杂的任务，可以将其拆分为以下的子任务：

(1) 找出一个人的收入。

(2) 计算总共有多少职员。

(3) 计算工资总额。

(4) 用职员人数去除工资总额。

计算工资总额本身又可分为一系列子任务：

(1) 找出每个职员的档案。

(2) 读出工资数额。

(3) 把工资加到部分和上。

(4) 读出下一个职员的档案。

类似地，读出每个职员档案中的记录又可以分解为一系列子任务：

(1) 打开职员的档案。

(2) 找出正确记录。

(3) 从存储设备中读取数据。

结构化程序设计成功地为处理复杂问题提供了有力的手段。然而到 20 世纪 80 年代末，它的一些缺点越来越突出。

当数据量增大时，数据与处理这些数据的方法之间的分离使程序变得越来越难以理解。



对数据处理能力的需求越强，这种分离所造成的负作用越显著。

采用结构化程序设计方法的程序员发现，每一种相对于老问题的新方法都要带来额外的开销，与可重用性相对，通常称之为重复投入。基于可重用性的思想是指建立一些具有已知特性的部件，在需要时可以插入到程序之中。这是一种模仿硬件组合方式的做法，当工程师需要一个新的晶体管时，他不用自己去发明，只要到仓库去找就行了。对于软件工程师来说，在面向对象程序设计出现之前，虽然市面上有些代码的功能看上去很像是自己需要的，但是修修改改最后还得自己动手重做。

1.4　面向对象程序设计

面向对象程序设计的本质是把数据和处理数据的过程当成一个整体——对象。

C++充分支持面向对象程序设计。面向对象程序设计的实现需要封装和数据隐藏技术，需要继承和多态性技术。

1. 封装和数据隐藏

当一个技术员要安装一台计算机时，他将各个设备组装起来。当他想要一个声卡时，不需要用原始的集成电路芯片和材料去制作一个声卡，而是购买一个他所需要的某种功能的声卡。技术员关心的是声卡的功能，并不关心声卡内部的工作原理。声卡是自成一体的。这种自成一体性称为封装性。无须知道封装单元内部是如何工作就能使用的思想称为数据隐藏。

声卡的所有属性都封装在声卡中，不会扩展到声卡之外。因为声卡的数据隐藏在该电路板上。技术员无须知道声卡的工作原理就能有效地使用它。

C++通过建立用户定义类型（类）支持封装性和数据隐藏。完好定义的类一旦建立，就可看成是完全封装的实体，可以作为一个整体单元使用。类的实际内部工作应当隐藏起来，使用完好定义的类的用户不需要知道类是如何工作的，只要知道如何使用它就行。

2. 继承和重用

要制造新的电视机，可以有两种选择：一种是从草图开始；另一种是对现有的型号加以改进。也许现有的型号已经令人满意，但如果再加一个功能，会更加完美。电视机工程师肯定不想从头开始，而是希望制造另一种新型电视机，该机是在原有的型号基础上增加一组电路做成的。新的电视机很快就制造出来了，被赋予一种新的型号，于是新型电视机就诞生了。这是继承和重用的实例。

C++采用继承支持重用的思想，程序可以在扩展现有类型的基础上声明新类型。新子类是从现有类型派生出来的，称为派生类。新型电视机是在原有型号的电视机上增加若干种功能而得到的，所以新型电视机是原有电视机的派生，继承了原有电视机的所有属性，并在此基础上增加了新的功能。

3. 多态性

通过继承的方法构造类，采用多态性为每个类指定表现行为。例如，学生类应该有一个计算成绩的操作。大学生继承了中学生，或者说是中学生的延伸。对于中学生，计算成绩的



操作表示语文、数学、英语等课程的成绩计算；而对于后继的大学生，计算成绩的操作表示高等数学、计算机、普通物理等课程的成绩计算。

继承性和多态性的组合，可以轻易地生成一系列虽类似但独一无二的对象。由于继承性，这些对象共享许多相似的特征。但由于多态性，一个对象可以有独特的表现方式，而另一个对象有另一种表现方式。

1.5 程序开发过程

大多数现代的编译程序都提供了一个集成开发环境。在这样一个环境中，一般是从菜单中选定 compile 或 make 或 build 命令，来生成可执行的计算机程序。

程序员编制的源程序被编译(compile)后，会生成一个目标文件，这个文件通常以.obj作为文件扩展名。该目标文件为源程序的目标代码，即机器语言指令。但这仍然不是一个可执行的程序，因为目标代码只是一个个的程序块，需要相互衔接成为一个适应一定的操作系统环境的程序整体。为了把它转换为可执行程序，必须进行连接(link)。

C++程序通常是通过同时连接一个或几个目标文件与一个或几个库而创建的。库(.lib)是一组由机器指令构成的程序代码，是可连接文件。库有标准库和用户生成的库。标准库是由 C++提供的，用户生成的库是由软件开发商或程序员提供的。文件与库连接的结果，即生成计算机可执行的程序。

程序员首先在集成开发环境中编辑源程序，或在其他编辑器中输入源程序，然后，在集成环境中启动编译程序将源程序转化成目标文件。编译之后，很有可能产生一些编译错误，于是程序员回到编辑状态重新开始编辑程序和编译。同样在紧接着的连接和运行中也会遇到连接或运行错误，此时，又回到编辑状态修改程序，见图 1-1。

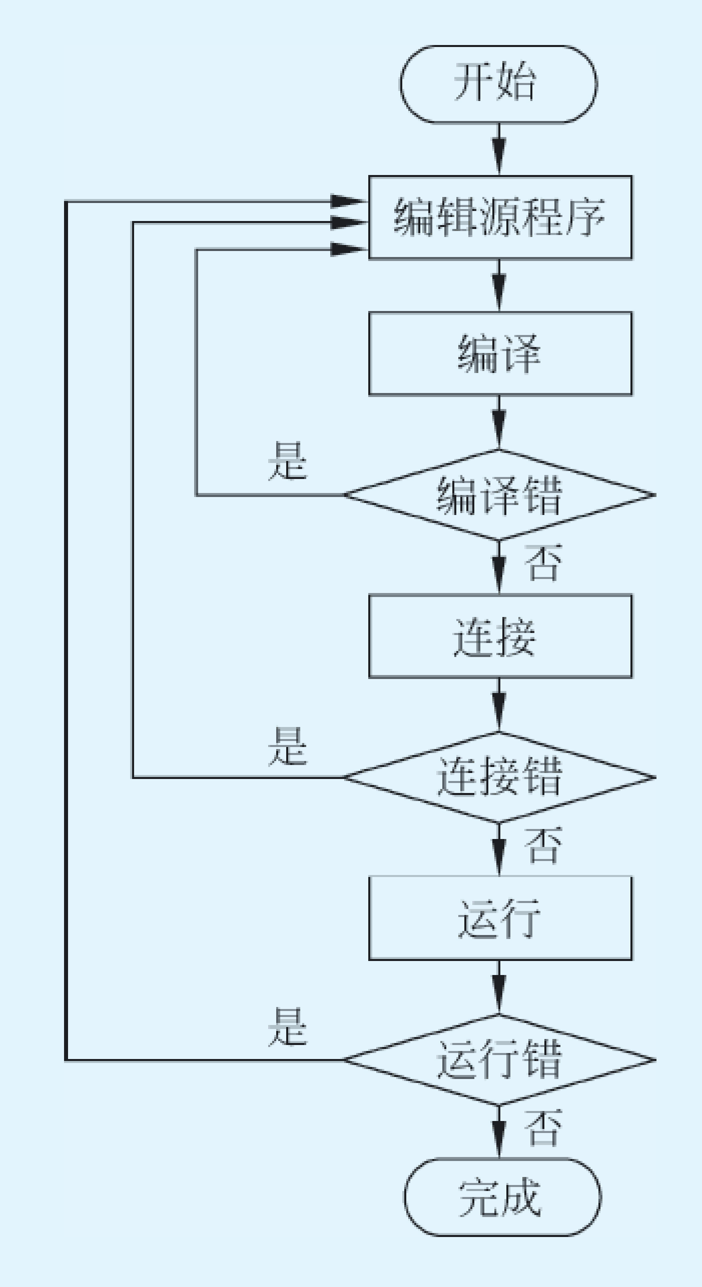
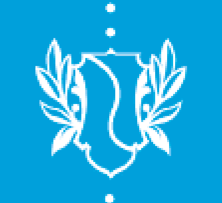


图 1-1　开发 C++程序的步骤



1.6 最简单的程序

我们从最简单的程序例子来分析 C++的程序构成。

```
//--------------------
// ch1_1.cpp
//--------------------
#include<iostream>
using namespace std;
int main()
{
  cout<<"I am a student.\n";
}//--------------------
```

运行结果为：

```
I am a student.
```

C++的程序结构由注释、编译预处理和程序主体组成。

注释是程序员为读者作的说明，是提高程序可读性的一种手段。一般可将其分为两种：序言注释和注解性注释。前者用于程序开头，说明程序或文件的名称、用途、编写时间、编写人以及输入输出说明等；后者用于程序中难懂的地方。

C++的注释为"//"之后的内容，直到换行。注释仅供阅读程序使用，是程序的可选部分。在生成可执行程序之前，**C++忽略注释，并把每个注释都视为一个空格**。

另外，C++还兼容了 C 语言的注释，即一对符号"/*"与"*/"之间的内容。它可以占多行，例如：

```
/* ----------------------------
    this is the simplest program.
   ---------------------------- */
```

每个以符号"#"开头的行，称为编译预处理行。如"#include"称为文件包含预处理命令。编译预处理是 C++组织程序的工具，有关内容在 6.8 节中介绍。

"#include <iostream>"的作用是在编译之前将文件"iostream"的内容增加(包含)到程序 ch1_1.cpp 中，以作为其一部分。iostream.h 是系统定义的一个"头文件"，它设置了 C++的 I/O 相关环境，定义输入输出流对象 cin 与 cout 等。cin 与 cout 的使用方法将在 2.6 节中介绍，其意义将在第 19 章中介绍。

main()表示主函数，每一个 C++程序都必须有一个 main()函数。main()作为从计算机操作系统进入(调用)程序的入口。main 前面的 int 表示函数的返回类型。既然 main()函数被操作系统调用，其最终也将返回到操作系统。main()函数用 int 作为返回类型是 C 和 C++的共同规定。函数体用大括号{}括起来。**描述一个函数所执行算法的过程称为函数定义**。例如，这里的 main()函数头和函数体构成了一个完整的函数定义。

函数名 main 全部都是由小写字母构成。C++程序中的名字是大小写"敏感"的，所以在书写标识符的时候要注意其大小写。

在 main()函数体中，cout(全是小写字母)是一个代表标准输出的流设备，它是 C++预

定义的对象(在 iostream 中定义),前面包含的头文件就是为了能在这里使用输出设备 cout。当程序要在设备上进行输出时,就需要在程序中指定该对象。输出操作由操作符“<<”来表达,它表示将该操作符右边的数据送到显示设备上。

程序中用双引号括起的数据“I am a student.\n”被称为字符串。其中字符“\n”表示一个回车控制符。字符串在 2.4 节中介绍。

“;”表示一个语句的结束。

例如,下面的程序求一个表达式的值:

```
//---------------------
// ch1_2.cpp
//---------------------
#include<iostream>
using namespace std;
//---------------------
int main(){
  int a,b,result;
  cout<<"please input two numbers:\n";
  cin>>a>>b;
  result=3*a-2*b+1;
  cout<<"result is "<<result<<endl;
}//---------------------
```

运行结果为:

```
please input two numbers:
123 45<ENTER>
result is 280
```

该程序从 main()开始运行。C++中,**一个变量必须在声明之后才能使用**,所以程序首先进行变量定义。“int a,b,result;”表示分别定义 a、b、result 这 3 个 int(整型)变量。C++语言提供的标准数据类型之一是 int。定义变量时,要求在变量之前声明变量的类型。**在C++中定义变量,意味着给变量分配内存空间**,用来存放变量值。

随后,在显示“please input two numbers:”之后,执行“cin>>a>>b;”,它从标准输入设备(键盘)中输入两个整型数 a 和 b。运行中,屏幕将等待输入,直至输入了两个数 123 和 45。输入时,两个数之间用空格隔开。这两个数分别赋给了变量 a 和 b。

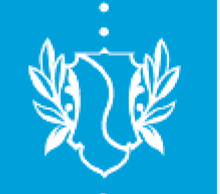
“result=3*a-2*b+1;”是赋值语句,* 是乘号,将表达式 3*a-2*b+1 的值(280)赋给变量 result,使之等于 280。然后,在接下来的语句中将 result 值输出。在 cout 语句中,有 3 个“<<”符号,表示各项内容的连续输出。“<<result”表示输出变量的值,“<<endl”表示输出一个回车符,与“<<'\n'”是等价的。

在输出格式中,<ENTER>表示输入的回车符,在以后的例子中将省略之。

1.7 函数

1. C++用函数组织程序

虽然 main()也是函数,但它并不是普通的函数。其他函数都是在程序运行时被调用。程



序命令按照它们在源代码中出现的顺序一句一句地顺序执行,直到碰到新的函数调用。然后程序调头去执行函数调用。当函数完成时,程序控制立即返回到调用函数的下一行代码。

这一过程可比喻为查字典。如果你在看书时有一个字不认识,你就要停止阅读,去查字典。字典查完后,再接着看书。

当程序需要服务时,它可以调用函数实现所需要的服务,然后当函数返回时再从它原来的地方继续执行。

2. C++程序是函数驱动的

例如,下面的程序实现一个简单的用户函数 max()的调用,来求两个数中的较大值,并调用了标准库函数 sqrt()来求两个数中较大值的平方根:

```
//---------------------
// ch1_3.cpp
//---------------------
#include<iostream>
#include<cmath>
using namespace std;
//---------------------
double max(double x, double y);
//---------------------
int main(){
  double a,b,c;
  cout<<"input two numbers:\n";
  cin>>a>>b;
  c=max(a,b);
  cout<<"the squart of maximum="<<sqrt(c);
}//--------------------
double max(double x, double y){
  if(x>y)
    return x;
  else
    return y;
}//--------------------
```

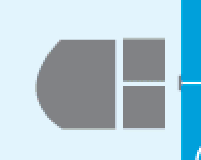
运行结果为:

```
input two numbers:
123 456
the squart of maximum=21.3542
```

主函数 main()的开始是 3 个 double 型(双精度类型)变量的定义语句,C++为此分配 3 个 double 型变量的内存空间。在输入了两个变量 a、b 的值(运行中输入的 123 赋给 a,456 赋给 b)后,调用了用户自定义的函数 max()。

C++中,**一个函数必须在函数声明后才能使用(被调用)**,所以在主函数 main()的前面,有 max()函数的声明。函数声明告诉编译器该函数是存在的。然后编译器在看到该函数被调用时就不会觉得大惊小怪了。同时编译器还对函数调用进行正确性检查。**C++函数声明总是由函数原型构成的**。函数原型在 5.2 节介绍。

max()函数调用使程序执行 max()函数中的语句,并将该函数的返回值赋给变量 c。



max()函数是求两个 double 型数中的大者,然后将结果返回给调用它的函数。所以在函数的头上写有 double 的返回类型。如果一个函数不需要返回值,则可以在头上声明为 void。

函数定义由函数头和函数体构成。函数头又由返回类型、函数名和函数参数构成。上例中的"double max(double x, double y)"就是函数头。函数体是由紧随函数头之后的大括号构成。

函数头中的函数参数允许向函数传递值。max()函数中,x、y 就是函数的参数。**参数声明时,要指出其类型**。函数 max()中的参数声明为"double x,double y",它指出 x 和 y 的类型都是双精度型。

函数定义中的参数称为形式参数,简称形参。函数 max()中的 x 和 y 就是形参。**调用函数时实际传递的值称为实际参数**,简称实参。主函数 main()在对 max()函数的调用时,用的 a 和 b 就是实参。函数在调用时,将实参值复制给形参,使得形参变量也具有实参的值。实参可以是表达式,它代表赋值的一方。形参只能是变量,因为它要接受赋值。

函数头有返回类型说明时,函数体中要用 return 返回值。同时,return 语句也使函数退出。max()函数中执行"return x"或"return y"即返回一个 double 值到主函数 main()中。

如果函数体中没有 return 语句,函数将在结尾处自动无值返回。如果有返回值,则该返回值应该具有函数头中声明的返回类型。

在 ANSI C++98 中,main 函数的返回类型规定为 int,但为了兼容老旧 C++,main 函数的返回类型可以是任何已有的数据类型,例如,编译器接受 void main(){}函数。

main()函数是唯一不是 void 返回类型,而可以在函数结束时忽略 return 语句的函数。因为只有这个函数不能被其他函数调用,仅由操作系统直接控制,其值返回给操作系统,在技术上独立实现返回过程,不影响 C/C++的函数返回机制。C++标准对 main()函数中不写 return 语句予以了默许。

函数有两种:标准库函数和用户定义函数。上例中的 max()函数是用户定义函数,sqrt()函数是标准库函数。标准库函数简称库函数,它是 C++提供的,可以为任何程序所使用。库函数无须用户声明和定义,但要将含有其函数声明的头文件包含在程序中。sqrt()函数的声明在 cmath 头文件中,所以在上例程序的开头写有#include<cmath>。

一般的数学函数,在 C 语言中,是放在头文件 math.h 中,而在 C++则是放在 cmath 头文件中,C++对 C 的函数库进行了些许改造。因为 C 与老旧的 C++的头文件都要带.h 后缀,而 C++又是兼容了 C,所以,ch1_3.cpp 代码中包含的头文件语句还可以写成#include <math.h>。

一个C++程序由一个主函数和若干个函数构成。由主函数调用其他函数,其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次,见图 1-2。

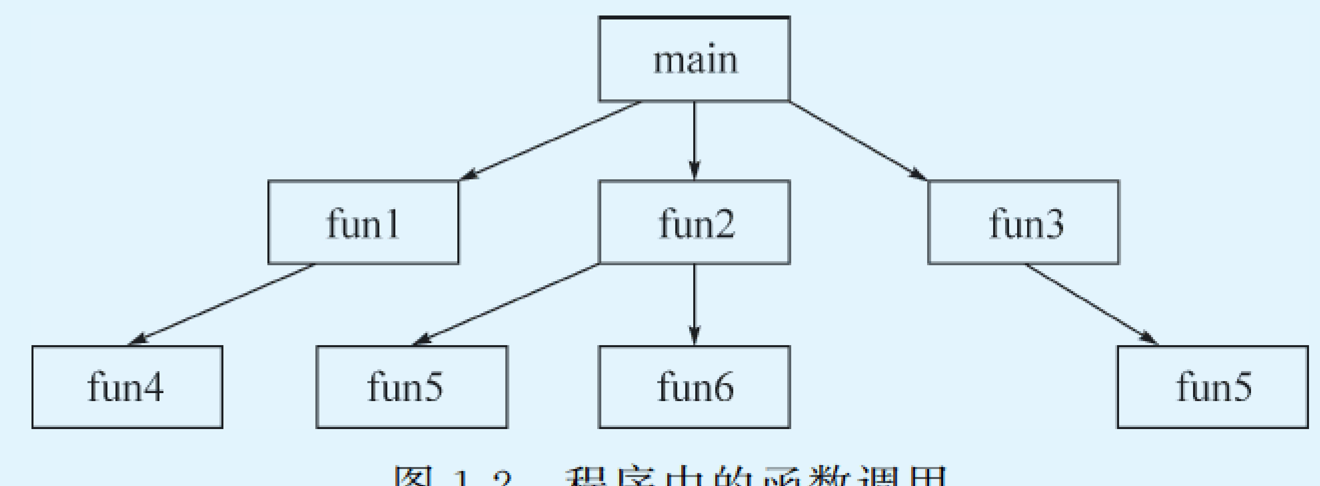


图 1-2 程序中的函数调用

函数定义包含函数声明,所以可以将函数定义放在函数应该声明的位置,而将其他函数



的定义放在主函数 main()之前。一个程序中主函数 main()的位置是没有特殊要求的,由此,程序 ch1_3. cpp 可以写成下面的程序:

```
//---------------------
//     ch1_4.cpp
//---------------------
#include<iostream>
#include<cmath>
using namespace std;
//---------------------
double max(double x,double y){ //既是函数定义又是函数声明
  if(x>y)
    return x;
  else
    return y;
}//---------------------
int main(){
  double a,b,c;
  cout<<"input two numbers:\n";
  cin>>a>>b;
  c = max(a,b);
  cout<<"the squart of maximum = "<<sqrt(c);
}//---------------------
```

该程序的功能和 ch1_3. cpp 是一样的。但通常我们习惯将主函数 main()放在程序的前面,以便阅读程序时很快找到。

C++中,每个函数对于程序的其他函数总是可见的。也就是说,任何函数都可以被包括它自己的所有函数所调用。用户不能定义函数的唯一之处是在另一个函数的定义之中。函数定义可以以任何顺序出现在程序中。由于 main()启动和终止程序运行,所以 main()函数通常第一个出现在程序中,而其他函数定义紧随其后。

小结

学习 C++,不一定非要学过 C,但学过 C 能促进 C++的学习。

C++程序经过编辑、编译和连接,产生可运行的 exe 文件。

C++程序由函数构成,它总是从主函数 main()开始运行。但并不是说,main()函数非得要写在程序的最前面。

函数有两种:标准库函数和用户定义函数。main()函数是特殊的用户定义函数。每个程序只能有一个 main()函数,并且必须要有一个 main()函数。

函数调用前必须要有函数声明。

函数定义包含函数声明。函数定义由函数头和函数体组成。关于函数,在第 5 章中将详细介绍。

一个语句可以写在多个程序行上,一个程序行可以写多个语句。语句以分号结束。

C++通过标准输入/输出流进行输入/输出。

程序 ch1_3. cpp 是C++的简单程序结构之样板。认识 C++程序从该程序开始。

程序设计的目标在正确的前提下,其重要性排列次序依次为可读、可维护、可移植和高效。

第2章　基本数据类型与输入/输出

程序中最基本的要素之一是数据类型。确定了数据类型，才能确定变量的空间大小和其上的操作。C++的数据类型检查与控制机制，奠定了C++今天的地位。C++还提供了I/O流机制，完成对输入/输出的操作管理。在过程化程序设计中，经常要碰到printf和scanf的输入/输出方式，它们是C++对C的兼容。学习本章后，要求搞清数据类型与变量、常量的关系，掌握各种常量的性质和定义，学会I/O流的使用，了解printf和scanf的作用。

2.1　字符集与保留字

每种语言都使用一组字符来构造有意义的语句。C++程序是用下列字符所组成的字符集写成的：

26个小写字母　　abcdefghijklmnopqrstuvwxyz
26个大写字母　　ABCDEFGHIJKLMNOPQRSTUVWXYZ
10个数字　　0123456789
其他符号　　+- * / = , . _ : ; ? \ " ' ~ | ! # % & ( ) [ ] { } ^
<>（空格）

C++中，保留字也称关键字。它是预先定义好的标识符，这些标识符对C++编译程序有着特殊的含义。表2-1列出了C++的保留字。ANSI C规定有32个保留字，表中用黑正体字表示；ANSI C++在此基础上补充了29个保留字，表中用黑斜体字表示。本书不作介绍的表中用白体字表示。为了使语言能更好地适应软件开发环境，BC或VC对保留字进行了扩充，在表中用白斜体字表示。BC与VC对关键字的扩充内容是不同的，这里只是常用的和共同扩充的几个。

表2-1　C++保留字

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**

int	**long**	register	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	union
unsigned	**void**	volatile	**while**
bool	***catch***	***class***	***const_cast***
delete	dynamic_cast	explicit	false
friend	***inline***	mutable	namespace
new	***operator***	***private***	***protected***
public	reinterpret_cast		static_cast
template	***this***	***throw***	true
try	typeid	typename	using
virtual	wchar_t		
asm	*cdecl*	*far*	*huge*
interrupt	*near*	*pascal*	*export*
except	*fastcall*	*saveregs*	*stdcall*
seg	*syscall*	*fortran*	*thread*

在程序中用到的其他名字(标识符)不能与 C/C++的关键字有相同的拼法和大小写。关键字也不能重新定义。

2.2 基本数据类型

一个程序要运行,就要先描述其算法。描述一个算法应先说明算法中要用的数据,数据以变量或常量的形式来描述。每个变量或常量都有数据类型。

变量是存储信息的单元,它对应于某个内存空间。用变量名代表其存储空间。程序能在变量中存储值和取出值。

在定义变量时,说明的变量名字和数据类型(如 int)告诉编译器要为变量分配多少内存空间,以及变量中要存储什么类型的值。

内存单元的单位是字节。如果要建立的变量类型的长度是两个字节,变量就需要保留两个字节的内存。变量的数据类型的作用之一是告诉编译器要为变量分配多少内存。

数据类型简称类型。在不同的计算机上,每个变量类型所占用的内存空间的长度不一定相同。例如,在 16 位计算机中,整型变量占两个字节,而在 32 位计算机中,整型变量占 4 个字节。C++的数据类型有基本数据类型和非基本数据类型之分。基本数据类型是 C++内部预先定义的数据类型,包括 char(字符型)、int(整型)、float(浮点型)和 double(双精度型)。非基本数据类型是基本数据类型的合成或用户定义数据类型,在 ANSI C++中,还有 wchar_t(双字节字符型)和 bool(布尔型)。

C++的数据类型见图 2-1,其中 type 表示非空数据类型。

除上述一些基本数据类型外,还有一些数据类型修饰符,它用来改变基本类型的意义,以便更准确地适应各种情况的需要。修饰符有 long(长型符)、short(短型符)、signed(有符号)和 unsigned(无符号)。

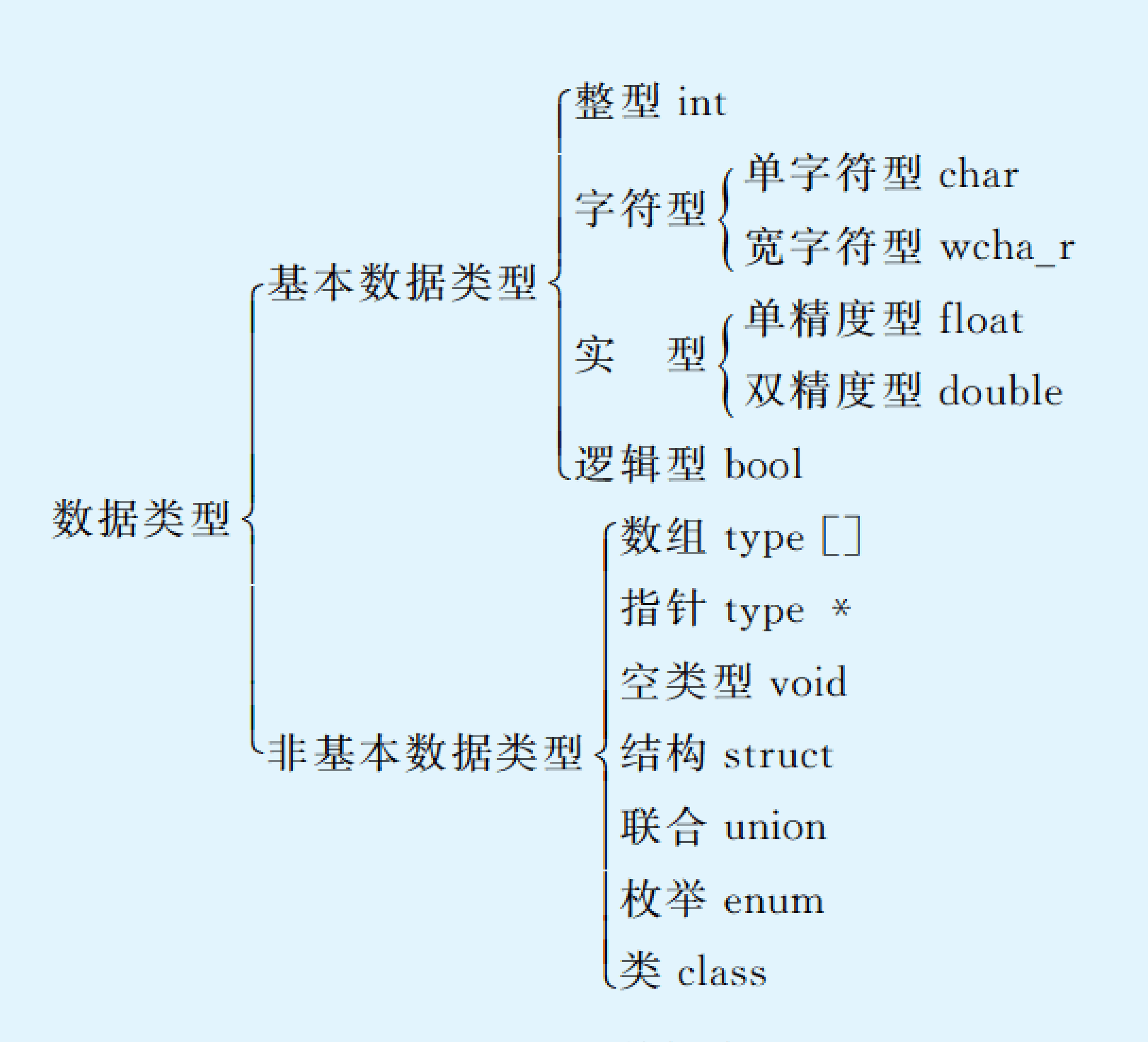
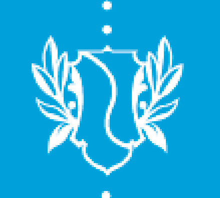


图 2-1　C++的数据类型

数据类型的描述确定了其内存所占空间大小，也确定了其表示范围。以在16位计算机中表示为例，基本数据类型加上修饰符的描述见表2-2。

表 2-2　常用基本数据类型描述

类　型	说　明	长度（字节）	表示范围	备　注
char	字符型	1	−128～127	$-2^7\sim(2^7-1)$
unsigned char	无符号字符型	1	0～255	$0\sim(2^8-1)$
signed char	有符号字符型	1	−128～127	$-2^7\sim(2^7-1)$
int	整型	4	−32 768～32 767	$-2^{15}\sim(2^{15}-1)$
unsigned int	无符号整型	4	0～65 535	$0\sim(2^{16}-1)$
signed int	有符号整型	4	−32 768～32 767	$-2^{15}\sim(2^{15}-1)$
short int	短整型	2	−32 768～32 767	$-2^{15}\sim(2^{15}-1)$
unsigned short int	无符号短整型	2	0～65 535	$0\sim(2^{16}-1)$
signed short int	有符号短整型	2	−32 768～32 767	$-2^{15}\sim(2^{15}-1)$
long int	长整型	4	−2 147 483 648～2 147 483 647	$-2^{31}\sim(2^{31}-1)$
signed long int	有符号长整型	4	−2 147 483 648～2 147 483 647	$-2^{31}\sim(2^{31}-1)$
long long int	长长整型	8	−9 223 372 036 854 775 808～9 223 372 036 854 775 807	$-2^{63}\sim(2^{63}-1)$
signed long long int	有符号长长整型	8	−9 223 372 036 854 775 808～9 223 372 036 854 775 807	$-2^{63}\sim(2^{63}-1)$
unsigned log long int	无符号长长整型	8	0～18 446 744 073 709 551 615	$0\sim(2^{64}-1)$
float	浮点型	4	$-3.4\times10^{38}\sim3.4\times10^{38}$	7位有效位
double	双精度型	8	$-1.8\times10^{308}\sim1.8\times10^{308}$	15位有效位
long double	长双精度型	10	$-1.2\times10^{4932}\sim1.2\times10^{4932}$	19位有效位



在大多数计算机上,short int 表示 2 字节长。short 只能修饰 int,short int 可以省略为 short。

long 只能修饰 int 和 double。修饰为 long int(可以省略为 long)时,一般表示 4 字节,修饰 long double 时,一般表示 10 字节。

unsigned 和 signed 只能修饰 char 和 int。一般情况下,默认的 char 和 int 为 signed。实型数 float 和 double 总是有符号的,不能用 unsigned 修饰。

用 sizeof(数据类型)可以确定某数据类型的字节长度。例如用下面的语句:

```
cout <<"size of int is " << sizeof(int) << endl;
```

在 16 位计算机上将输出:

```
size of int is 4
```

C++是强类型语言。强类型语言要求程序设计者在使用数据之前对数据的类型明确声明,不能默认。例如,在存储一个整数值之前,首先必须告诉计算机要存储的是一个整数类型。使用强类型语言有很多好处,例如,错把一个整数当成一个职工的编号,编译器就会产生错误信息提示。

强类型语言是通过编译器的功能来体现的。**一个编译器能检查出的错误越多,我们就说该编译器越好**。

在程序设计中,绝大部分的错误是发生在数据类型的误用上,所以现代程序设计语言都要求是强类型语言,因它能够检查出尽可能多的数据类型方面的错误。

例如,如果一个圆半径用浮点数(float)表示,占用 4 字节的内存,而一个短整数只占用 2 字节的内存。如果要清除一个圆半径值,那么就要清除 4 字节的内容。如果把一个短整数当作一个圆半径来清除,那么除了清除短整数的 2 字节外,还清除了其他 2 字节的内容。也许额外清除的 2 字节中包含重要的信息,这种错误的清除会造成意想不到的后果。

对于一个初学者来说,那种对数据类型要求不严格的语言(如 BASIC)或许更易于编程。但是,它不会像强类型语言那样能够捕获错误,因而调试和运行的程序可能出现更多的错误。

2.3 变量定义

1. 命名变量名

C++程序是大小写敏感的,即大写和小写字母认为是不同的字母。例如变量名 something、Something、SOMETHING 和 SomeThing 都是不同的名字。

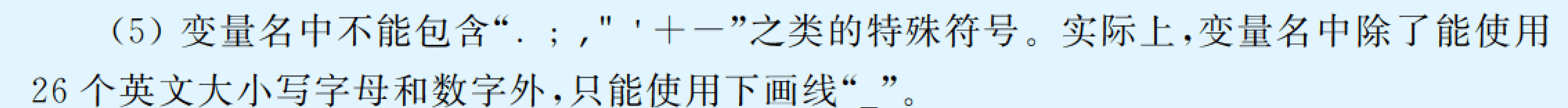
给变量命名要遵守如下的规则:

(1) 不能是 C++关键字。

(2) 第一个字符必须是字母或下画线。

(3) 不要太长,一般不超过 31 个字符为宜(太长则书写困难,反而不美)。

(4) 中间不能有空格。



(5) 变量名中不能包含“. ; ," '+-”之类的特殊符号。实际上,变量名中除了能使用26个英文大小写字母和数字外,只能使用下画线“_”。

(6) 变量名不要与C++中的库函数名、类名和对象名相同。

例如,下面是一些变量名:

```
way_cool, RightOn, Bits32, Case, iPtr, myCar          //合法
case, 52Select, A Lot, -VV                            //非法
sin, cout, string                                     //不合适
```

变量通常具有描述性的名称。例如,看到numberOfStudents这个变量名,就立刻可以想到它表示学生人数,即使简写成numOfStudent也是一目了然。而D6Xy就不是一个好名称,猜不透它代表一个序列号、一个商标,还是一种机器名。给变量命名的方式决定了程序书写的风格。在整个程序中保持同一风格很重要。

许多程序员喜欢变量名全用小写字母。如果名字需要两个单词(如my car),常用的命名有两种:my_car和myCar。后一种形式称为骆驼表示法,因为大写字母看起来像驼峰。

有些人觉得my_car较可读,也有人因为它难以输入而避免使用下画线。本书使用骆驼表示法,即以小写字母开头,后部的单词都以大写字母开头,如myBook、theFox、sizeOfChar。

自定义的类型名(如类和结构类型)则以大写字母开头。一些函数名也以大写字母开头。

还有一种特别流行的方法是匈牙利标记法(Hungarian notation),该方法在每个变量名的前面加上若干表示类型的字符,如iMyCar表示整型变量,ipMyCar表示整型指针等。这种方法已经流行于现代软件开发环境中,如Windows中的类库和函数库。

2. 变量定义方式

可以在一个语句中建立多个同一类型的变量,方法是在类型后写上多个变量名,中间用逗号隔开。例如:

```
unsigned int myAge, myWeight;    //2个无符号整型变量
long int area, width, length;    //3个长整型变量
```

在同一语句中不能混合定义不同类型的变量。

3. 变量赋值与初始化

用赋值运算符“=”给变量赋值。例如:

```
unsigned short width;
width = 5;          //赋初值
```

也可以在定义时直接给变量赋值。在定义的同时,赋给变量一个初始值,称为变量的初始化。例如:

```
unsigned short width = 5;    //定义并初始化
```

对于整型变量来说,赋初值的形式用两条语句完成,初始化的形式只用一条语句完成。



它们都是先给变量分配一个存放整数的内存空间,然后将一个整数值赋给该变量。其初始化与赋初值的效果完全一样。然而当定义常量或定义一个对象时,二者的差别则很大。

在定义时也可以初始化多个变量。例如:

```
long width = 7, length = 7;
```

不是所有的变量在定义时都需要初始化。例如:

```
double area, radius = 23;
```

该变量定义并不是将 23 同时赋给这两个变量,而是变量 radius 初始化为 23,area 只是定义,没有被初始化。

4. typedef

用 typedef 可以为一个已有的类型名提供一个同义词。用法是:以 typedef 开始,随后是要表示的类型,最后是新的类型名和分号。例如:

```
typedef double profit;        //定义 double 的同义词
typedef int INT, integer;     //定义两个同义词
INT a;                        //即 int a;
profit d;                     //即 double d;
```

typedef 没有实际地定义一个新的数据类型,在建立一个 typedef 类型时没有分配内存空间,typedef 在程序中起到帮助理解的作用。

2.4 字面量

1. 整型数

整型数即整型字面量。整型数可以有 3 种表示方式:

(1) 十进制整数。如 123、−456、0。

(2) 八进制整数。以 0 开头的整数是八进制数。如 0123 表示八进制数$(123)_8$,等于十进制数 83。

(3) 十六进制数。以 0X 或 0x 开头的数是十六进制数。如 0X123 或 0x123 表示十六进制数$(123)_{16}$,等于十进制数 291。

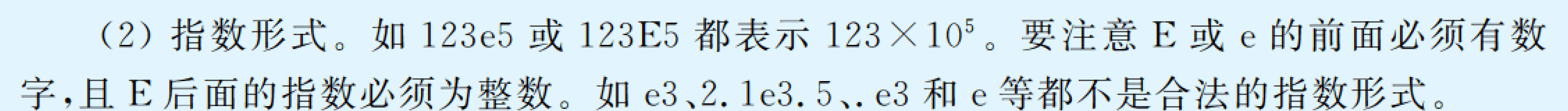
C++中,十进制数有正负之分,但八进制数和十六进制数只能表示无符号整数。所以,若写成−011 或−0X345,C++并不能将其理解为负数。

如果在整型数后面加一个字母 L 或 l,则认为是 long int 型整数。例如 123L 是 long int 型整数。

2. 实型数

实型数即实型字面量。实数在 C++中就是浮点数。实数有两种表示方式:

(1) 小数形式。它由数字和小数点组成(注意必须有小数点)。如 0.123、.234、0.0 等都是十进制数。



（2）指数形式。如 123e5 或 123E5 都表示 123×10^5。要注意 E 或 e 的前面必须有数字，且 E 后面的指数必须为整数。如 e3、2.1e3.5、.e3 和 e 等都不是合法的指数形式。

实型数分为单精度(float)、双精度(double)和长双精度(long double)3 类。一般情况下，float 型数据在内存中占 4 字节，double 型数据占 8 字节，long double 型数据占 10 字节。float 型提供 7 位有效数字，double 型提供 15 位有效数字，long double 型提供 19 位有效数字。

在 C++ 中，一个实型数如果没有任何说明，表示 double 型；要表示 float 型数，则必须在实数后加上 f 或 F；要表示 long double 型数，则必须在实数后加 l 或 L。例如：

```
34.5f         //float 型
34.5          //double 型(默认表示)
34.5L         //long double 型
34.5l         //long double 型
34.5e23f      //float 型
34.5e23       //double 型(默认表示)
34.5e23L      //long double 型
34.5e23l      //long double 型
34.5e400      //long double 型(因为范围超过 double 表示)
```

3. 字符

字符是用单引号括起来的一个字符。如'a'、'x'、'?'、'$'等都是字符。

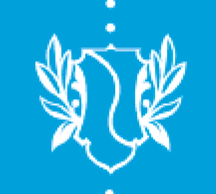
除了以上形式的字符外，C++ 中还允许使用一种特殊形式的字符，即以“\”开头的字符序列。如'\n'，它代表一个换行符。表 2-3 列出了 C++ 中常用的以“\”开头的特殊字符。

表 2-3　C++ 常用特殊字符

字符形式	值	功　能
\a	0x07	响铃
\n	0x0A	换行
\t	0x09	制表符(横向跳格)
\v	0x0B	竖向跳格
\b	0x08	退格
\r	0x0D	回车
\\	0x5C	反斜杠字符“\”
\"	0x22	双引号
\'	0x27	单引号
\ddd		1～3 位八进制数
\xhh		1～2 位十六进制数

表中列出的字符称为转义字符，意思是将反斜杠“\”后面的字符转变成另外的意义。有些是控制字符，如“\n”；有些是表示字符的符号，如“'”。

反斜杠可以和八进制数或十六进制数值结合起来使用，以表示相应于该数值的 ASCII 码值。例如，'\03'表示 Ctrl-C，'\0A'表示回车。转义字符使用八进制数表示时，最多是 3 位数，不必以 0 开头，如'\03'等价于'\3'。转义字符的八进制数表示的范围是'\000'～'\377'，即从 $0\sim255_{10}$。转义字符使用十六进制数表示时，是 2 位数，用 x 或 X 引导，表示的范围



是'\x00'～'\xff'。

例如,下面的代码响铃输出一个字符串:

```
cout <<"\x07operating\tsystem\n";
```

其输出内容为在响铃的同时显示:

```
operating       system
```

该结果中两个单词之间的空隙是制表符'\t'产生的作用。

字符变量定义的形式为:

```
char c1,c2;
```

它表示c1、c2为字符型变量,各只能放一个字符。初始化字符变量可以用下述形式:

```
char c1 = '\n', c2 = '\x07', c3 = 'B', c4 = '\xff', c5 = 97;
```

将一个字符赋值给字符变量,实际上并不是把该字符本身放到内存单元中,而是将该字符的相应ASCII码(整型数)存入。例如,字符'a'的ASCII码是97,上例中"c5=97"即为"c5='a'"。

在内存中,字符数据以ASCII码存储,即以整数表示,所以C++中字符数据和整型数据之间可以相互赋值,只要注意其表示的范围合理即可。例如:

```
int a = 'b';     //ok:   给一个整型变量赋一个字符值
char c = 97;     //ok:   给一个字符变量赋一个整型值
```

字符数据和整型数据在输出中的表示是不同的。例如,对上面的赋值,输出:

```
cout << a << endl;
cout << c << endl;
```

其结果为:

```
98
a
```

尽管整型变量a赋给了字符值,字符变量c赋给了整型值,但它们在内存中都以整数的形式表示。在输出时,a是整型,所以就按整型数的表示方式输出;c是字符型,所以就按字符的表示方式输出。

'0'与0是截然不同的两个数。'0'是数字字符,其ASCII码等于值48或0x30。而0则是整数值。除此之外,'\0'和NULL也表示整数0。

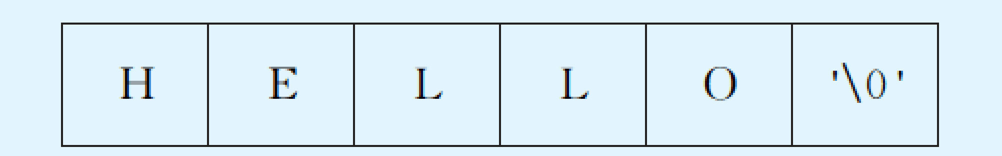
4. 字符串

字符串是由一对双引号括起来的字符序列。例如:

```
"How do you do?"
"I am a student."
"hello"
```

都是字符串。字符和字符串是不同的。在C++中,字符串总是以'\0'结束。如果有一个字符

串"HELLO",那它在内存中的表示为连续 6 个内存单元,见图 2-2。



H	E	L	L	O	'\0'

图 2-2　字符串的内存表示

不能将字符串赋给字符变量。例如:

```
char c = "abc";    //error: 不能将字符串转换成字符型
```

一个字符占有一个内存单元,含有一个字符的字符串占有两个内存单元,第 2 个内存单元存放 0 结束符。所以,**"0"与'0'是不同的**。在 8.7 节将进一步介绍字符串。我们将看到,字符串实际上是字符指针类型。所以,字符与字符串类型不同,占用内存大小不同,处理方式不同,决定了它们的使用方式也不同。

单个字符的字符串与一个字符在输出的表示上没有差别,因为字符串输出时,C++并不把 0 结束符一起输出。例如:

```
cout <<"a" << endl;
cout <<'a' << endl;
```

输出结果为:

```
a
a
```

5. 枚举符

枚举符可以通过建立枚举类型来定义。

定义枚举类型的语法是先写关键字 enum,后跟类型名、大括号,大括号括起来的是用逗号隔开的每个枚举符,最后以分号结束定义。例如:

```
enum COLOR{ RED, BLUE, GREEN, WHITE, BLACK };
```

例中,COLOR 是枚举类型名,**它不是变量名,所以不占内存空间**。枚举类型定义规定枚举类型名和枚举的取值范围。

枚举符是一种符号常量。RED、BLUE 等是符号常量,它们表示各个枚举值,在内存中表示以整型数。如果没有专门指定,第 1 个符号常量的枚举值就是 0,其他枚举值依次为 1 往上加。所以,C++自动给 RED 赋以 0,BLUE 赋以 1,等等。

可以给枚举符指定枚举值,也可以部分指定枚举值。例如:

```
enum COLOR{ RED = 100, BLUE = 200, GREEN, WHITE = 400};
```

例中,GREEN 没有被赋给值,便被自动赋以值 201。

定义了枚举类型后,可以定义该枚举类型的变量。变量的取值只能取枚举类型定义时规定的值。例如:

```
COLOR paint = GREEN;
```

该例定义了一个枚举变量 paint,用枚举符 GREEN 所代表的枚举值初始化。



不能用整数值赋给枚举变量。例如：

```
paint = 200;    //error
```

在 VC 中会得到一个“不能将整常数赋给枚举变量”(cannot convert from 'const int' to 'enum paint')的错误。在 BC 中会得到一个“将整数赋给枚举变量 paint”的警告。BC 的警告比 VC 显得缓和一些，但都表示整数赋给枚举型变量是不合适的。**任何警告对于程序设计都应引起与编译错误同等的重视**。

2.5 常量

常量是常数或代表固定不变值(字面值)的名字。

程序中如果想让变量的内容自初始化后一直保持不变，可以定义一个常量。

例如，在圆面积计算中经常要用常数 π，此时，通过命名一个容易理解和记忆的名字来改进程序的可读性，同时在定义中加关键字 const，给它规定为常量性质，以帮助预防程序出错。

如果在整个程序中的许多地方都要用到一个字面值，那么在这些地方的一个或多个地方写错了，这个值就会导致计算错误。如果给字面值取个名字，每处都以该名字代替，那么编译器就能检查名字拼写错误，避免字面值的不一致性。

π 字符不属于 C++语言的字符描述集，不能用来作 C++中的名字。我们用 pi 来表示 π：

```
const float pi = 3.1415926;
```

由于有效位的限制，在下面的常量定义中，最后 3 位不起作用：

```
const float pi = 3.141592653;
```

尽管等号后面的字面值是 double 型的，但因为 float 常量只能存储 7 位有效位精度的实数，所以 pi 的实际值为 3.141593(最后 1 位四舍五入)。如果将常量 pi 的类型改为 double 型，则能全部接受上述 10 位数字。

定义成 const 后的常量，程序中对其只能读不能修改，从而可以防止该值被无意地修改。由于不可修改，不能赋值，所以，常量定义时必须初始化。例如：

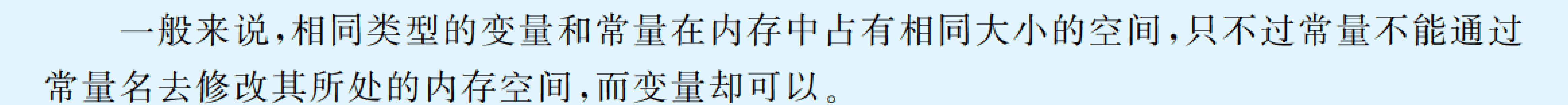
```
const float pi;    //error
pi = 3.1415926;    //error
```

常量名不能放在赋值语句的左边。

常量定义中初始化的值可以是一个不依赖于运行的表达式。该表达式在程序运行之前就能计算，所以，编译时就能求值。例如：

```
const int size = 100 * sizeof(int);       //ok
const int number = max(15,23);            //error
```

因为 sizeof 不是函数，而是 C++的基本操作符，该表达式的值在编译之前能确定，所以第一个常量定义语句合法。第二个语句要求函数值，函数一般都要在程序开始运行时才能求值，该表达式不能在编译之前确定其值，所以是错误的。



一般来说，相同类型的变量和常量在内存中占有相同大小的空间，只不过常量不能通过常量名去修改其所处的内存空间，而变量却可以。

关于#define：

在C中，另一种定义常量的方法是用编译预定义指令(#define)。例如：

```
#define PI 3.1415926
```

这条语句的格式是#define后面跟一个常量名再跟一串字符，中间用空格隔开。由于它不是C++语句，所以行末不用分号。

当程序被编译时，它要先被编译预处理。当预处理遇到#define指令时，就用数值3.1415926替换程序中所有的PI。

尽管它具有常量的所有属性，但是在编译预处理完成后，PI就不属于C++程序中的名字了(全部被字串3.1415926所替代)，所以它不是一个具有一定类型的常量名。随后的编译无法发现由它引起的数据类型误用的错误。

C++容许#define定义常量是为了兼容C，使C程序能在C++编译器中顺利通过。在C++编程中，常量定义都用const，不用#define。

2.6 I/O流控制

1. I/O的书写格式

I/O流是输入或输出的一系列字节，当程序需要在屏幕上显示输出时，可以使用插入操作符“<<”向cout输出流中插入字符。例如：

```
cout <<"This is a program.\n";
```

当程序需要执行键盘输入时，可以使用抽取操作符“>>”从cin输入流中抽取字符。例如：

```
int myAge;
cin >> myAge;
```

不管把什么基本数据类型的名字或值传给流，它都能懂。

例如，下面的函数输出字符串和整数：

```
#include <iostream>
using namespace std;
int main()
{
    cout <<"My name is Jone\n";
    cout <<"the ID is ";
    cout << 2;
    cout << endl;
}
```

上面的输出也可以在同一行中串连，下面的输出语句与上例输出同样内容：

```
cout <<"My Name is Jone\n" <<"the ID is " << 2 << endl;
```

也可以分在几行,提高可读性,下例语句与上例输出同样结果:

```
cout <<"My Name is Jone"       //行末无分号
     <<"the ID is "
     << 2
     << endl;
```

cin 可以用和 cout 一样的方式调整行,它自动识别变量位置和类型。例如:

```
int i; float f; long l;
cin >> i >> f >> l;
```

cin 能够知道抽取的变量的类型,它将对 i、f、l 分别给出一个整型数、浮点型数和长整型数。

2. 使用控制符

流的默认格式输出有时不能满足特殊要求,例如:

```
double average = 9.400067;
cout << average << endl;
```

希望显示的是 9.40,即保留两位小数,可是却显示了 9.40007;默认显示 6 位有效位。

用控制符(manipulators)可以对 I/O 流的格式进行控制。控制符是在头文件 iomanip 中定义的对象。可以直接将控制符插入流中。常用控制符如表 2-4 所示。

表 2-4 I/O 流的常用控制符

控制符	描述
dec	置基数为 10
hex	置基数为 16
oct	置基数为 8
setfill(c)	设填充字符为 c
setprecision(n)	设显示小数精度为 n 位
setw(n)	设域宽为 n 个字符
fixed	固定的浮点显示
scientific	指数表示
left	左对齐
right	右对齐
skipws	忽略前导空白
uppercase	十六进制数大写输出
lowercase	十六进制数小写输出

使用控制符时,要在程序的头上加头文件#include<iomanip>。

3. 控制浮点数值显示

使用 setprecision(n)可控制输出流显示浮点数的数字个数。C++默认的流输出数值有效位数是 6。

如果 setprecision(n)与 fixed 合用,可以控制小数点右边的数字个数。fixed 是设置定



点小数表示法。

如果与 scientific 合用，可以控制指数表示法的小数位数。scientific 是设置指数方式的小数表示法。

例如，下面的代码分别用浮点、定点和指数方式表示一个实数：

```
//---------------------
//    ch2_1.cpp
//---------------------
#include<iostream>
#include<iomanip>       //要用到 setprecision()
using namespace std;
//---------------------
int main(){
  double amount = 22.0/7;
  cout << amount << endl;
  cout << setprecision(0)<< amount << endl
      << setprecision(1)<< amount << endl
      << setprecision(2)<< amount << endl
      << setprecision(3)<< amount << endl
      << setprecision(4)<< amount << endl;

  cout << fixed << setprecision(8)<< amount << endl;
  cout << scientific << amount << endl;
  cout << setprecision(6); //重新设置成原默认设置
}//---------------------
```

运行结果为：

```
3.14286
3
3
3.1
3.14
3.143
3.14285714
3.14285714e+00
```

该程序在 32 位机器上运行通过。

在普通表示的输出中，setprecision(n)表示有效位数。

第 1 行输出数值之前没有设置有效位数，所以用流的有效位数默认设置值 6。第 2 行输出设置了有效位数 0，C++最小的有效位数为 1，所以作为有效位数设置为 1 来看待。第 3～6 行输出按设置的有效位数输出。

在确定表示的输出中，setprecision(n)表示小数位数。

第 7 行输出是与 fixed 合用，所以 setprecision(8)设置的是小数点后面的位数，而非全部数字个数。

在指数形式输出时，setprecision(n)表示小数位数。

第 8 行输出用 scientific 来表示指数表示的输出形式。其有效位数沿用上次的设置值 8。



小数位数截短显示时,进行四舍五入处理。

4. 设置值的输出宽度

除了使用空格来强行控制输出间隔外,还可以用 setw(n)控制符。如果一个值的显示数需要比 setw(n)确定的字符数 n 更多,则该值将显示所有字符。例如:

```
float amount = 3.14159;
cout << setw(4) << amount << endl;
```

其运行结果为 3.14159。它并不按 4 位宽度输出,而是按实际宽度输出。

如果一个值的字符数比 setw(n)确定的字符个数更少,则在数字字符前显示空白。不同于其他控制符,setw(n)仅仅影响下一个数值输出,换句话说,使用 setw 设置的间隔方式并不保留其效力。例如:

```
cout << setw(8)
     << 10
     << 20 << endl;
```

运行结果为:

```
______1020
```

运行结果中的下横线表示空格。整数 20 并没有按宽度 8 输出。setw()的默认值为宽度 0,即 setw(0),意思是按输出数值的表示宽度输出,所以 20 就紧挨着 10 了。若要每个数值都有宽度 8,则每个值都要如下设置:

```
cout << setw(8) << 10
     << setw(8) << 20 << endl;
```

5. 输出八进制数和十六进制数

3 个常用的控制符是 hex、oct 和 dec,它们分别对应十六进制数、八进制数和十进制数的显示。这 3 个控制符在 iostream 头文件中定义。例如:

```
//---------------------
// ch2_2.cpp
//---------------------
#include<iostream>
using namespace std;
//---------------------
int main(){
  int number = 1001;
  cout <<"Decimal:" << dec << number << endl
       <<"Hexadecimal:"<< hex << number << endl
       <<"Octal:" << oct << number << endl;
}//--------------------
```

运行结果为:

```
Decimal:1001
Hexadecimal:3e9
Octal:1751
```

1001 是一个十进制数，不能把它理解成十六进制数或八进制数，因为它不是以 0x 或 0 开头。但在输出时，流根据控制符进行解释操作，使其按一定的进制来显示。

用 uppercase 可以控制十六进制数大写输出。例如，在上例中对十六进制数进行大写控制，即：

```
#include<iostream>
using namespace std;

//… …

cout<<"Hexadecimal:"<<hex<<uppercase<<number<<endl;
```

便能得到十六进制数的大写表示：Hexadecimal:3E9。

6. 设置填充字符

setw 可以用来确定显示的宽度。默认时，流使用空格符来保证字符间的正确间隔。用 setfill 控制符可以确定 setw 所规定的间隔字符。setfill 在头文件 iomanip 中定义。例如：

```
//---------------------
//    ch2_3.cpp
//---------------------
#include<iostream>
#include<iomanip>         //要用到填空设置、宽度设置
using namespace std;
//---------------------
int main(){
  cout<<setfill('*')
      <<setw(2)<<21<<endl
      <<setw(3)<<21<<endl
      <<setw(4)<<21<<endl;
  cout<<setfill(' ');    // 恢复默认设置
}//---------------------
```

运行结果为：

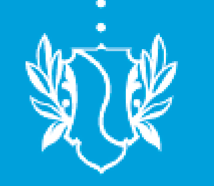
```
21
*21
**21
```

7. 左右对齐输出

默认时，I/O 流左对齐字串，右对齐数值。使用 left 和 right 标志，可以控制输出对齐。例如：

```
//---------------------
//    ch2_4.cpp
//---------------------
#include<iostream>
#include<iomanip>      //要用到 setw()
```






```
using namespace std;
//----------------------
int main(){
  cout << right
      << setw(5)<< 1
      << setw(5)<< 2
      << setw(5)<< 3 << endl;
  cout << left
      << setw(5)<< 1
      << setw(5)<< 2
      << setw(5)<< 3 << endl;
}//--------------------
```

运行结果为：

```
    1    2    3
1    2    3
```

8. 强制显示小数点和符号

当程序输出下面的代码时：

```
cout << 10.0/5 << endl;
```

默认的 I/O 流会简单地显示 2，而非 2.0，因为除法的结果是精确的。当需要显示小数点时，可以用 showpoint 标志。例如：

```
//----------------------
//     ch2_5.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
int main(){
  cout << 10.0/5 << endl;
  cout << showpoint << 10.0/5 << endl;
}//--------------------
```

运行结果为：

```
2
2.00000
```

默认时，I/O 流仅在负数之前显示值的符号，根据程序的用途，有时也需要在正数之前加上正号，可以用 showpos 标志。例如：

```
//----------------------
//     ch2_6.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
int main(){
  cout << 10 <<" "<< -20 << endl;
  cout << showpos << 10 <<" "<< -20 << endl;
}//--------------------
```

运行结果为：

```
10   -20
 +10   -20
```

2.7 printf 与 scanf

printf 和 scanf 是标准输入/输出函数。它们是 C 程序中所使用的，在头文件 stdio.h 中声明了这两个函数。在 C++面向对象程序设计中，I/O 流代替了它们(见第 19 章)。在过程化程序设计中，printf 和 scanf 在使用习惯上可作为 C++流的一个补充。

1. printf 函数

1) printf 函数的一般格式

```
printf(格式控制字符串,输出项 1,输出项 2, ...)
```

括号中的格式控制字符串和输出项都是函数参数。printf()函数的功能是将后面的参数按给定的格式输出。

格式控制字符串中有格式说明也有普通字符。格式说明由“%”和格式字符组成，如%d、%f 等。它的作用是将输出的数据转换成指定的格式输出。普通字符就是原样输出的字符。输出项 n 是需要输出的一些数据，可以是表达式。例如：

```
#include <stdio.h>
void f()
{
   int a = 10,b = 20;
   printf("%d,%d",a,b);
}
```

在上例中，双引号中的字符除了两个“%d”外，还有逗号字符“,”，它是普通字符。a、b 的值分别为 10、20，输出为：

```
10,20
```

2) %d 格式符

%d 用来输出十进制整数，可以有长度修饰。例如：

```
int a = 28,b = 38;
long c = 289868;
printf("%5d,%5d\n%ld\n",a,b,c);
printf("%3ld\n%7ld\n%d\n",c,c,c);
```

输出结果为：

```
28,   38
289868
289868
289868
289868
```

其中,%5d 表示输出宽度为 5;%ld 表示输出为长整型;%3ld 表示输出宽度为 3 的长整型数,如果长整型数的位数多于 3,则按长整型的位数输出。如果一个长整型数只按整型数输出,则会引起整数截断,即输出长整型数低位 2 个字节的值。上例中最后一个数的输出就是截断长整型数 289868 得到的输出。时下的绝大多数编译器都是 32 位的,即整型数在内部用 32 位表示,长整型就是整型,%ld 与%d 效果一样,因而最后的输出不会进行截断。

3) %o 和%x 格式符

%o 和%x 用来以八进制数和十六进制数输出。八进制数和十六进制数都是无符号整数,输出时不带符号。例如:

```
int a=-3;
printf("%d,%o,%x,%X,%6x\n",a,a,a,a,a);
```

输出结果为:

```
-3,37777777775,fffffffd,FFFFFFFD,fffffffd
```

当用%X 时,输出十六进制数时用大写字母;当用%x 时,输出用小写字母。

八进制数和十六进制数同样可以用"%lx"输出长整型数,另外也可以指定输出字段的宽度。

4) %u 格式符

%u 用来以无符号十进制整数方式输出。有符号整数(int 型)可以%u 格式输出,无符号整数(unsigned 型)可以%d 格式输出或以%o、%x 格式输出。另外,还可以指定格式宽度。例如:

```
unsigned int a=65533;
int b=-2;
printf("a=%d,%o,%x,%u,%7u\n",a,a,a,a,a);
printf("b=%d,%o,%x,%u,%6u\n",b,b,b,b,b);
```

运行结果为:

```
a=65533,177775,fffd,65533,65533
b=-2,37777777776,fffffffe,4294967294,4294967294
```

5) %c 格式符

%c 用来以字符方式输出。如果一个整数的值在 0～255,也可以字符方式输出。它们都可以指定格式宽度。例如:

```
char ch='a';
int a=65;
printf("%c,%d,%3c\n",ch,ch,ch);
printf("%c,%d,%3d\n",a,a,a);
```

输出结果为:

```
a,97,  a
A,65, 65
```

6) %s 格式符

%s 用来以字符串格式输出。可以指定格式宽度。如果字符串长小于指定的宽度时,

可以选择左对齐和右对齐。另外还可以选择字符串的前 n 个字符。例如：

```
printf("%s","Hello\n");
printf("Hello\n");
printf("%3s,%-5.3s,%5.2s\n","Hello","Hello","Hello");
```

输出结果为：

```
Hello
Hello
Hello,Hel  ,   He
```

输出字符串时，要求 printf()的第 2 个参数是字符串，但不一定是字符串常量。我们将在第 8 章中介绍字符串变量的概念。

如果输出的只有一个字符串，可以省略格式参数，如上例中第二条语句，因为格式参数本身可以是原样输出的普通字符串。

“%−5.3s”中的负号表示左对齐，如果没有负号，则默认为右对齐。5 表示格式宽度，3 表示截取字符串中 3 个字符。

7）%f 格式符

%f 用来以小数方式输出。可以指定格式宽度，也可以指定小数位数，还可以规定左对齐和右对齐。例如：

```
float x=123.456;
double y=321.654321;
long double z=3.141592653;
printf("%f,%-7.2f,%10.4f\n",x,x,x);
printf("%lf,%-7.2lf,%10.4lf\n",y,y,y);
printf("%Lf,%-7.2Lf,%10.4Lf,%14.10Lf\n",z,z,z,z);
```

输出结果为：

```
123.456001,123.46,   123.4560
321.654321,321.65,   321.6543
3.141593,3.14,     3.1416,   3.1415926530
```

以“%f”格式输出时，默认的小数位数为 6，所以输出 123.456001。x 值的有效位是 7 位，如果超过了 7 位，就不能保证其精度了，所以输出结果并不是想象中的 123.456000。“%−7.2f”表示左对齐，总长度 7 位，小数位数 2 位。

“%lf”是 double 型输出，有效位是 15 位。“%Lf”是 long double 型输出。由于默认小数位数是 6，所以看到的输出中，小数位数只有 6 位。

8）%e 格式符

%e 用来以指数方式输出浮点数。指数的输出格式要求小数点前必须有且只有一位非 0 数字，默认的小数位数为 6，一般 float 和 double 默认的指数位数为 2(不包括 e+或e−)，long double 默认的指数位数为 3。例如：

```
float x=123.456;
double y=321.654321;
long double z=3.141592653e+123;
printf("%e,%-7.2E,%10.4e\n",x,x,x);
printf("%le,%-7.2lE,%10.4le\n",y,y,y);
printf("%Le,%-7.2LE,%10.4Le,%14.10Le\n",z,z,z,z);
```



输出结果为：

```
1.234560e+02,3.22E+02,1.2346e+01
3.216543e+02,3.22E+02,3.2165e+02
3.141593e+123,3.14E+123,3.1416e+123,3.1415926530e+123
```

其中，e 或 E 控制指数形式的 e 以小写或大写方式输出。

此外，还有%g 格式符，用以输出浮点数，它根据数值的大小，自动选取 f 格式或 e 格式(较短的一种)。

如果要输出%本身，则双写%。例如：

```
printf("%f%%",1.0/3);
```

输出结果为：

```
0.333333%
```

2. scanf 函数

scanf 的一般形式为：

```
scanf(格式控制字符串,地址 1,地址 2, ...);
```

格式控制字符串的含义同前，地址 n 是变量的地址。

%d 用以输入整数，可以带 l 表示长整数，带 h 表示短整数。

%c 用以输入字符。

%o、%x 用以输入八进制数和十六进制数。%lo 和%lx 分别表示长八进制数和长十六进制数。

%f 用以输入浮点数，%lf 和%Lf 分别表示输入 double 型数和 long double 型数。

%e 与%f 作用相同。

%s 用以输入字符串，以非空字符开始，以空字符或回车结束。

例如：

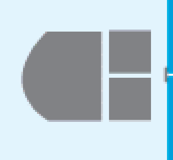
```
int a,b;
char ch1,ch2;
float f,g;
scanf("%d %d",&a,&b);
scanf("%c%c",&ch1,&ch2);
scanf("%f,%f",&f,&g);
```

输入时：

```
23 456
ab
23.6712,612.97
```

两个输入项之间一般用空格分开。如果规定了分隔字符，如逗号“,”，则必须以逗号分开。如上例中，第 1 个 scanf 语句数值之间必须以空格或回车隔开，第 3 个 scanf 语句数值之间必须以逗号隔开。

在用“%c”格式输入时，空格字符和转义字符都作为有效字符输入。如上例中，执行第



2 个 scanf 语句时，若输入“a b”，则 'a' 赋给 ch1，' '(空格)赋给 ch2，而 'b' 被忽略。

scanf()中后面的地址参数是重要的，不能是变量名，否则会将输入值存放在变量值作为地址的内存空间中，导致意想不到的运行异常。

小结

变量是程序分配给某个内存位置的名字，它可以存放信息。程序在使用变量前，必须先说明变量名和变量类型。

不同的变量不能同名，变量名应该尽量反映出变量的用途，以增强程序的可读性。

在程序运行中，常量的值不可改变。常量也有各种数据类型，也占有存储空间。各种数据类型的数据表示有一定的范围，越过了该范围，C++就要对该数据进行截取，使得数据不再正确。

利用 cout 可以输出各种数据类型的数据，可以多种方式在屏幕上显示输出信息(包括特殊符号)。

C++兼容 C 的库函数，所以 printf()和 scanf()也可照常使用。

练习

2.1 设整数 42 486，请定义一个变量，初始化之，并以八进制与十六进制数输出。如果将该整数定义成无符号短整数，当以有符号数输出时，结果是什么？请用补码概念解释。

2.2 取圆周率为 3.141 592 6，分别输入半径为 40 和 928.335，求圆面积，要求各数据按域宽 10 位输出，先输出圆周率和半径，再输出其面积。

2.3 将常数 e(2.718 281 828)作为常量定义，然后输出其 10 位有效位数的浮点数、定点方式和 8 位小数位表示的数，以及指数形式和 8 位小数位表示的数。

2.4 设学生常数为 500，编程输出下列结果(引号也要输出)。

```
"How many students here?"
"500"
```

2.5 用 sizeof 操作符求出表 2-2 中各数据类型的字节长度，并按：

```
size of char        1 byte
size of int         2 byte
...
```

的格式打印输出。

2.6 读下列程序。

(1) 运行时，输入 6、6、8 三个数，写出运行结果。

(2) 解释该程序做了什么，程序的书写为什么要分成几段，各起什么作用。

(3) 用 cout 和 cin 代替 printf()和 scanf()函数，改写程序(注意格式)。

(4) 将求面积部分的程序以调用函数的方式来完成，则函数声明、定义和调用作如何修改？

```
//----------------------
// exercise6
//----------------------
#include<stdio.h>
#include<math.h>
//----------------------
int main(){
  float a, b, c, s, area;
  printf("please input 3 sides of one triangle:\n");
  scanf("%f, %f, %f", &a, &b, &c);

  s=(a+b+c)/2;
  area=sqrt(s*(s-a)*(s-b)*(s-c));
  printf("a=%7.2f, b=%7.2f, c=%7.2f\n", a, b, c);
  printf("area of triangle is %10.5f\n", area);
}//---------------------
```

2.7　读下列程序,写出运行结果。

```
//----------------------
// exercise7
//----------------------
#include<iostream>
#include<conio.h>      //需要调用屏幕输入 getche()
using namespace std;
//----------------------
int add(int x, int W);
//----------------------
int main(){
  cout<<"In main():\n";
  int a, b, c;
  cout<<"Enter two numbers:\n";
  cin>>a>>b;
  cout<<"\nCalling add():\n";
  c = add(a,b);
  cout<<"\nBack in main():\n";
  cout<<"c was set to "<<c<<endl;
  cout<<"\nExiting. . . \n";
  getche();
}//---------------------
int add(int x, int y){
  cout<<"In add(),received "<<x<<" and "<<y<<endl;
  cout<<"and return "<<x+y<<endl;
  return x+y;
}//---------------------
```

2.8　以函数调用的方式,求圆柱体的体积。

在主函数中先输入圆柱体的半径和高,然后调用求体积的函数,最后输出结果。

第3章　表达式和语句

程序是一些按次序执行的语句。执行语句是为了完成某个操作，修改某个数据。程序中大部分的语句是由表达式构成的，因为表达式直截了当地返回值。正因为如此，表达式是C++编译器处理的重要内容。学习本章后，要求理解表达式和语句的概念，掌握表达式中各种运算符的功能与特点，更好地理解 C++语言的强大与灵活。

3.1　表达式

1. 表达式概述

表达式是操作符、操作数和标点符号组成的序列，其目的是说明一个计算过程。

表达式可以嵌套，例如 2＋3＋(5 * sizeof(int))/345。

表达式根据某些约定、求值次序、结合性和优先级规则来进行计算。

所谓约定，即类型转换的约定。例如：

```
float a;
a = 5/2;
```

结果 a 得到值为 2。5/2 是整数除法取整，因为 5 和 2 都是整数，不会由于 a 是 float 型而轻易改变运算的性质。

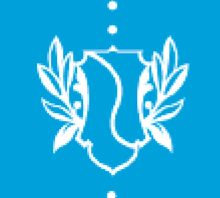
所谓求值次序，是指以正确计算表达式值为目的，以内部优化为手段，为每个操作数规定一个计值的顺序。求值次序视编译器不同而不同，见 3.9 节。

所谓结合性，是指表达式中出现同等优先级的操作符时，该先做哪个操作的规定。例如：

```
d = a + b - c;           //C++规定,加减法先左后右。先做 a + b,其结果再减去 c
d = a = 3;               //C++规定,等号是先右后左。先做 a = 3,其结果再赋给 d
```

所谓优先级，是指不同优先级的操作符，总是先做优先级高的操作。例如：

```
d = a + b * c;           //乘法优先级比加法高。先做 b * c,其结果再与 a 相加
```



2. 左值和右值

左值(left value,缩写为 lvalue)是能出现在赋值表达式左边的表达式。左值表达式具有存放数据的空间,并且存放是允许的。例如:

```
int a = 3;            //a 是变量,所以 a 是左值
const int b = 4;      //b 是常量,所以 b 不是左值
```

显然常量不是左值,因为 C++规定常量的值一旦确定是不能更改的。

右值(right value,缩写为 rvalue)只能出现在赋值表达式的右边。左值表达式也可以作为右值表达式。例如:

```
int a,b = 6;
a = b;              //b 是变量,所以是左值,此处作为右值
a = 8;              //8 是常量,只能作右值,不能作为左值
```

表达式终能产生数值结果,可表示左值或右值。例如:

```
int a;
(a = 4) = 28;              //ok:a = 4 是左值表达式,可以被赋以值 28
void f(){return ;}         //此为函数定义,该函数不返回任何值
```

28 是右值表达式,而 a=4 是左值表达式(C++的语法规定),所以可以放在赋值语句的左边。该语句表示 a 的值用 28 替代刚刚赋给的值 4。

函数定义 f()本身不是表达式,它说明了一个不返回值的函数。对函数 f()的调用是语句,它实施了一个没有返回值的操作。

3. 优先级和结合性

表 3-1 对操作符的优先级和结合性作了小结。表中包含了 C++所有的操作符,共有 16 级优先级。表中的操作符如重复出现,则第 1 次出现的是单目运算符,第 2 次出现的是双目运算符。每一级的结合性,不是从左到右就是从右到左。表达式中,在没有括号的情况下,这些规则决定了表达式运算的次序。

每一级中的操作符是同优先级的。

表 3-1 C++操作符的优先级与结合性

优先级	操　作　符	结合性
1	()、[]、-> ::	左→右
2	!、~、+、-、++ 、-- 、&、* (强制转换类型) sizeof new delete	右→左
3	. 、* 、->、*	左→右
4	* 、/、%	左→右
5	+、-	左→右
6	<<、>>	左→右
7	<、<=、>=、>	左→右
8	==、!=	左→右
9	&	左→右
10	^	左→右
11	|	左→右

续表

优先级	操　作　符	结合性
12	&&	左→右
13	||	左→右
14	?:	右→左
15	=、*=、/=、+=、-=、|=、<<=、>>=	右→左
16	,	左→右

4. 语句与块

C++中所有的操作运算都通过表达式来实现。由表达式组成的语句称为表达式语句，它由一个表达式后接一个分号";"组成。

通过计算表达式即执行了表达式语句。大多数表达式语句为赋值语句和函数调用。

语句是用来规定程序执行的控制流。在没有跳转和分支(见第 4 章)的情况下，语句将按照其在源程序中出现的次序顺序执行。

语句可以是空语句。空语句是只有一个分号而没有表达式的语句，其形式为：

```
;
```

它不产生任何操作运算，只作为形式上的语句，被填充在控制结构中。例如：

```
if(x>9)
  ;
else
   cout <<"not larger than 9\n";
```

例中判断 x 是否大于 9，如果大于 9，不做任何事，否则输出"not larger than 9"和回车。

块(或称复合语句)是指括在一对大括号{}里的语句序列。从语法上来说，块可以被认为是单个语句。例如：

```
if(x>9)
{
   cout <<"The number is perfect.\n";
   cout <<"It is larger than 9\n";
}
else
{
   cout <<"not larger than 9\n";
}
```

x 若大于 9，则执行两条输出语句，否则，输出"not larger than 9"和回车。这两条执行语句必须放在大括号中，因为 if 与 else 之间只能容纳一条语句，或一个语句块。而 else 后面的大括号则可以省略。此外，块还可以嵌套。

3.2 算术运算和赋值

1. 操作符种类

C++提供了算术运算符+、-、*、/、%。

+、-、* 是通常意义的加、减、乘法。

/对于整型数则为除法取整操作。例如,5/2 得到结果为 2。

/对于浮点数则为通常意义的除法。例如,5.0/2.0 得到结果为 2.5。

由此可见,/操作符可以对不同的数据类型进行不同的操作。事实上,+、-、* 、/、%对不同数据类型的操作都不同。如整数加法是将两个整型数相加,而浮点数加法是将两个浮点数相加,相加的具体操作(在机器指令级上)浮点和整数是不同的。

%只能对整型数进行操作。其操作意义为取余。例如,5%2 得到结果为 1。

%如果对浮点数操作,则会引起编译错误。

2. 赋值缩写

算术表达式的赋值表示为:

```
int x,y,z;
x = y * z;
x = y/z;
x = y + z;
x = y - z;
x = y % z;
```

表 3-1 中优先级为 15 的操作符都是赋值操作符。

当一变量出现在紧贴赋值操作符两边时,可以缩写。例如:

```
x = x * y;     缩写为:     x *= y;
x = x + y;     缩写为:     x += y;
x = x - y;     缩写为:     x -= y;
x = x/y;       缩写为:     x/= y;
x = x % y;     缩写为:     x %= y;
```

紧贴赋值操作符右边的变量必须与右值余部整体分离。例如:

```
x = x * 3 + y;不能写为:x *= 3 + y;
x = x * (3 + y);可以写为:x *= 3 + y;
```

赋值以及缩写都要求左边的表达式为左值,即 x 为左值。

赋值构成一个表达式,因而它具有值。赋值表达式的值为赋值符左边表达式的值。例如:

```
cout <<(x = 5) << endl;
```

将输出 5。同时 x 被赋予值 5。

赋值表达式为左值。例如:

```
(x = max(5,7)) += 3;
```

该语句先将 max(5,7)函数调用的值赋给 x,然后在此基础上增值 3。它等价于:

```
x = max(5,7) + 3;
```

缩写格式通常更有效,可读性也不差。缩写与不缩写的差别在于缩写式中左边的变量只出现一次,而不缩写的式中出现两次。例如:



```
(x = max(5,7)) = (x = max(5,7)) + 3;
```

上式是合法的表达式，它与前面讨论的表达式的不同之处在于 max(5,7)调用了两次。如果函数有副作用(见 3.9 节)，则赋值与赋值缩写是不同的。

3. 溢出

进行算术运算时，很可能溢出结果。发生溢出是由于一个变量被赋予一个超出其数据类型表示范围的数值。数值溢出是不会引起编译错误的，只要分母不为 0 也不会引起除 0 运行故障，但会使运行结果发生偏差。

例如，在 16 位机器上进行下面的操作：

```
int weight = 42896;
```

在 16 位机器中将不能得到值 42 896，而是－22 640。因为有符号整数的表示范围是－32 768～32 767，所以它只能得到 42 896 的补码－22 640(42 896－65 536)。

一个整数类型的变量，用任何一个超过表示范围的整数初始化，得到的值为用该整数范围作模运算后的值。例如：

```
int weight = 142896;
```

则当 weight 是 2 字节整型数时，得到值为 11 824。因为 142 896＝2×65 536 ＋11 824。而 142 896－3×65 536＝－53 712，该数不在有符号整型数表示范围内。

3.3 算术类型转换

C++遇到两种不同数据类型的数值进行运算时，会将两个数作适当的类型转换，然后再进行运算。转换的方向见图 3-1。

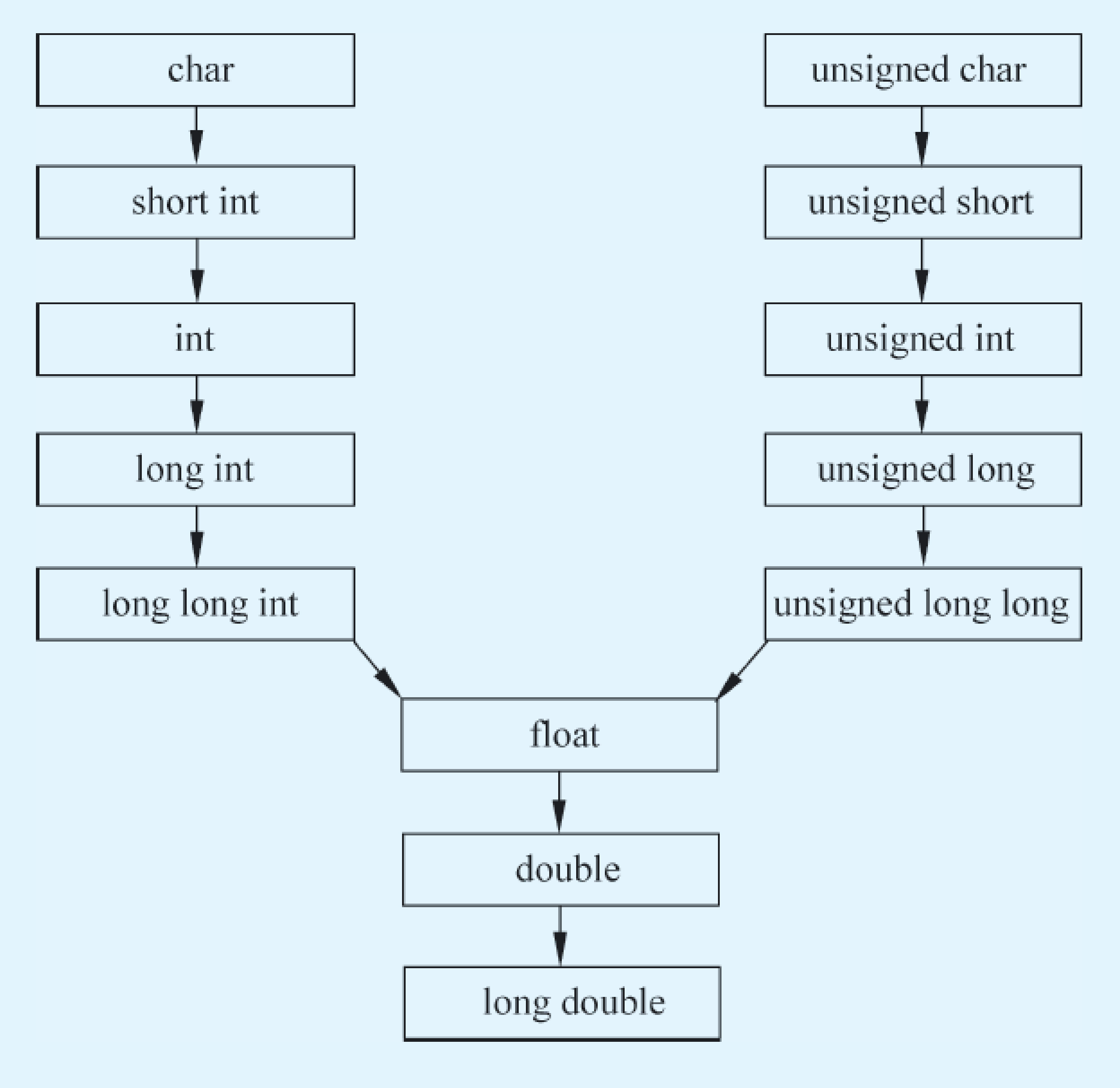


图 3-1　类型转换的方向



如果一个 char 型数和一个 int 型数相加，则先将 char 型数转换成 int 型数，然后进行运算。因为在图 3-1 中，char 有向 int 转换的趋势。如果一个 long int 型数和一个 float 型数相加，则先将 long int 转换成 float 型数，然后进行运算。如果一个 int 型数和一个 unsigned long 型数相乘，则先将 int 型数转换成 unsigned long 型数，然后进行运算。

转换总是朝表达数据能力更强的方向，并且转换总是逐个运算符进行的。例如：

```
float f = 3.5;
int n = 6;
long k = 21;
double ss = f * n + k/2;
```

ss 将会得到结果 31。计算 ss 时，首先将 f(float 型)和 n(int 型)转换成 float 型数，算得 21，然后计算 k/2 得整除运算结果 10(long int 型)，再将 long int 型的数字 10 转换成 float 型数。21 和 10 两个数相加，得到最后结果 31。

数据运算过程中自动进行的类型转换称为隐式类型转换。上例的表达式运算过程中进行的数据类型转换就是隐式转换。

有时候，我们会面临下面计算结果不准确的问题：

```
long m = 234 * 456/6;
```

即发现 m 为－4061，而不是 17 745。原因是语句先进行 int 型数的乘法运算，结果仍以 int 型数保留起来：234×456＝106 704＝2×65 536－24 368，取模之后得到－24 368。该数参加整除运算：－24 368/6 得－4061(取整)。由于中间的结果被截断，所以，最后的结果是错的。如果让第 1 次乘法的结果以 long 型数保留下来，就能得到正确的结果。这就要求参加乘法运算的两个数至少有一个为 long 型数。

例如，将其中之一标识以 L 或 l(long)，则可保证其正确：

```
cout << 234 * 456L/6 << endl;
```

输出结果为：

```
17784
```

还可以将整型数强制转换为 long 型：

```
cout <<(long)234 * 456/6 << endl;
```

该语句使 234 成为 long 型数，与整数 456 相乘，先隐式转换，再相乘，得到一个 long 型数 106 704。再与 6 相除取整，从而得到正确结果。

强制转换又称显式转换，其语法是在一个数值或变量前加上带括号的类型名。也可以类型名后跟带括号的数值或表达式。如上面语句也可以写成：

```
cout << long(234) * 456/6 << endl;
```

如果类型名是带类型修饰的，则要给类型名加括号。例如：

```
cout <<(unsigned long)234 * 456/6 << endl;
cout << unsigned long(234) * 456/6 << endl; //error
```

请注意下面语句不能产生所期望的效果：

```
cout << long(234 * 456)/6 << endl;
```

该语句首先执行括号里的乘法,得到一个 int 型整数(已被取模)−4061,然后强制转换为 long 型数,再参加除 6 取整运算,所以得不到正确结果。

3.4 增量和减量

增量和减量操作符表示为:++和--。

增量操作表示加 1,减量操作表示减 1。例如:

```
a++;    //等价于 a=a+1;
++a;    //等价于 a=a+1;
a--;    //等价于 a=a-1;
--a;    //等价于 a=a-1;
```

增量操作符有前增量与后增量之分。前增量操作++a 的意义为:先修改操作数使之增 1,然后将增 1 过的 a 值作为表达式的值。而后增量操作 a++的意义为:先将变量 a 的值作为表达式的值确定下来,再将 a 增 1。对于增量和减量操作符,它要求操作数是左值,因为操作数的值要发生变化。例如:

```
int a=3;
int b= ++a;          //相当于 a=a+1; b=a;
cout << a <<"     " << b << endl;
int c=a++;           //相当于 c=a; a=a+1;
cout << a <<"   " << c << endl;
```

输出的结果为:

```
4  4
5  4
```

b 被赋予了 4,因为前增量操作先将 a 自增为 4,然后作为表达式赋值。C 被赋予了 4,因为后增量操作使表达式的值(a)先赋给 c,然后 a 再自增为 5。

由于前增量操作返回的值即修改后的变量值,所以返回的仍是一个左值。例如:

```
int a=3;
++(++a);     //ok: ++a 是左值,可以接着做 ++ 操作
```

例中得到的 a 的值为 5。

由于后增量操作返回的值是原先 a 的数值,而后 a 的实体值已经发生变化,故返回的不能是当前的 a 值,只能是过去的 a 数值,不能是左值。例如:

```
int a=3;
++(a++);     //error: a++ 不是左值
```

相应的,有前减量--a 和后减量 a--。例如:

```
int a=3;
int b=--a;        //相当于 a=a-1; b=a;
cout << a <<"   " << b << endl;
int c=a--;        //相当于 c=a; a=a-1;
cout << a <<"   " << c << endl;
```



输出的结果为：

```
2  2
1  2
```

由于增量与减量操作修改内存实体，所以操作数不能是常量，它必须是一个左值表达式。例如：

```
3++;     //error
```

增量与减量操作符是两个＋或两个－的一个整体，中间不能有空格。如果有多于两个＋或两个－连写的情况，则编译首先识别前面两个＋或－为增量或减量操作符。

例如，对于“int a=1,b=5,c;”的变量定义，下面5个表达式，有些不允许：

```
c=a+b;                    //ok:c=6
c=a++b;                   //error:编译接收为 a ++ b
c=a+++b;                  //ok:编译接收为 a ++ + b
c=a++++b;                 //error:编译接收为 a ++ ++ b
c=a+++++b;                //error:编译接收为 a ++ ++ + b
```

第2行中，编译将其理解为a++b。由于++操作是单目运算符，所以该表达式语法错误。若要合法，应写成a++b，表示a加上正b。

第3行中，编译将其理解为a ++ ++b。同样由于++是单目操作符，引起编译错误。若要合法，应写成a+++ + b，表示a++加上正b。

第4行中，编译将其理解为a ++ ++ +b。由于a++是个非左值表达式，所以中间的++操作符是非法的。若要合法，应写成a+++ ++ b或者a++ + ++ b，表示a++加上++b。

3.5 关系与逻辑运算

1. 运算符

关系操作符有比较(＝＝)、大于(＞)、小于(＜)、大于等于(＞＝)、小于等于(＜＝)和不等于(！＝)。

逻辑运算符有非(!)、逻辑与(&&)和逻辑或(||)。

对于＞＝、＜＝、!＝、＝＝、&&、||都是一个操作符的整体，所以中间不能有空格，而且前3个操作符中的字符次序不能颠倒。例如，下面的写法都是不合法的：

```
=<, =>, =!,<  =,>  =,=  =,!  =
```

2. 比较运算符

比较(＝＝)和赋值(＝)是两个不同的操作，所以用的操作符也不同。比较用于测试给定的两个操作数是否相等。例如：

```
if(x==999)
   cout<<"x is 999\n";
```

C++中，表达式都产生值，赋值操作符产生的值正是所赋的值，而比较操作符产生的值


是比较的结果，可能是 0 或 1，即假或真。

真和假是逻辑值。在 C++ 中，假意味着 0，真意味着非 0。所以，任意一个非 0 数都是真，表示为逻辑值就是 1。例如：

```
x = somevalue;
if(x = 9)
   cout <<"x is not 0\n";
```

例中，不管 x 的初值是什么，总是执行 cout 语句。因为 x=9 是赋值表达式，其表达式的值是所赋的值 9，而 9 为非 0 值，所以 if 语句的条件为真，所以总是执行 cout 语句。又如：

```
x = somevalue;
if(x = 0)
   cout <<"x is 0\n";
```

例中，不管 x 以前是什么值，总是不会执行 cout 语句。因为 x=0 是赋值表达式，并且其值为 0，为假。

这一般不是编程的本意，但由于**=与==经常不小心搞错**，使得程序不正确地运行。在 BC 和 VC 编译器中，在像 if 语句这样的条件表达式中，遇到=时都会给予警告。这时，就应该有所警觉，以免程序错误地执行。

3. 不等于运算符

当要测试一些东西不是真时，可以使用不等于操作符。例如，如果要在一些东西是真时在屏幕上显示一则消息，则可以用如下语句：

```
if(x!= 9)
   cout <<"x isn't 9\n";
```

要注意的是，如果颠倒！=，则意义完全不同：

```
if(x = !9)
   cout <<"x isn't 9\n";
```

该 if 条件表达式是一个赋值语句，!9 为非真，即 0。所以该条件表达式相当于 if(x=0)，于是 cout 语句永远也不会执行。

4. 嵌入赋值

有时候，需要将一个函数值赋给一个变量，然后比较该变量的值与预定值是否相等。例如：

```
x = func();
if(x == somevalue)
   //语句
```

上面的代码与下面的代码等价：

```
if((x = func()) == somevalue)
   //语句
```

因为要给 x 赋值，然后确定 x 的值，所以先进行赋值。而赋值表达式的值即 x 的值可



以作为比较的操作数。这里,由于==操作比=操作优先级高,所以需要额外加一个括号。

这种赋值紧接着比较的表达式称为嵌入赋值,它经常在程序样例中看到。

5. 逻辑非运算符

! 改变条件表达式的真假值,即逻辑运算的"非"。原来是0,则 !0 为1;原来为非0,则 ! 操作使之变为0。例如:

```
if(!(x==9))
   cout <<"x is not 9.\n";
```

因为==比 ! 优先级低,所以额外的括号是需要的。包围"x==9"的括号使比较先进行,然后再做非操作。

6. 逻辑运算

&& 和||是两个逻辑运算符,它们的意义为求两个条件表达式的逻辑与和逻辑或。

例如,下面的代码为根据室温打印一则消息:

```
int temp=90,humi=80;

if(temp>=80 && humi>=50)
   cout <<"wow,it's hot!\n";
if(temp<60)||temp>80)
   cout <<"the room is uncomfortable.\n";
```

输出结果为:

```
wow,it's hot!
the room is uncomfortable.
```

因为 && 比>=优先级低,所以 if 中的条件表达式为先求 temp>=80 和 humi>=50,然后进行 &&(逻辑与)运算。

7. 短路表达式

如果多个表达式用 && 连接,则一个假表达式将使整个连接都为假(此处需要数理逻辑知识)。例如:

```
int n=3,m=6;

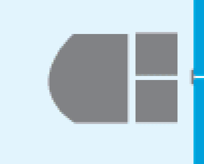
if(n>4 && m++<10)
   cout <<"m should not be changed.\n";

cout <<"m=  " << m << endl;
```

输出结果为:

```
m= 6
```

由于 n>4 的比较值为0,所以整个 if 条件表达式的值不用看后面就知道为0。C++利



用这个特点以产生高效的代码。所以，后面的表达式不被执行。这样，m 的值还是 6 而不是 7。知道了短路表达式在 C++中的处理方式，就可以在编写程序时，不但避免不必要的错误，而且还可利用它。例如：

```
if(b!= 0 && a/b>2)
    //语句
```

if 条件表达式中的 b!=0 若成立，才会执行后面的关系运算，做分母是 b 的除法。否则，跳过整个条件语句。

→ 在程序中，如果碰到除 0 运算，则运行发生异常。如果没有定义异常处理(见第 21 章)，则整个程序终止运行。

同理，如果多个表达式用||连接，则一个真表达式将使整个连接都为真。例如：

```
if(temp < 60||temp>80)
    cout <<"the room is uncomfortable.\n";
```

例中如果 temp<60 成立，则不会进行 temp>80 的关系比较，直接执行输出语句。

3.6 if 语句

1. if 语句

if 语句的语法为：

```
if(条件表达式)
    语句;
```

或：

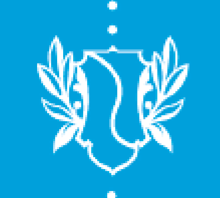
```
if(条件表达式)
{
    语句;
}
```

它的意义为：如果条件表达式进行一次测试，且测试为真，则执行后面的语句。C++中的 if 语句与其他计算机语言的 if 语句区别不大。

如果 if 语句只控制一条语句，则包围该语句的大括号不是必需的。

例如，下面的程序等待输入一字符，如果是'b'，则响铃：

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
    cout <<"please input the b key to hear a bell.\n";
    char ch = getche();
    if (ch == 'b')
        cout <<'\a';
}
```



2. 空语句

编译器必须在 if 条件表达式的后面找到一个作为语句结束符的分号";",以标志 if 语句的结束。这样,如果是下面的代码:

```
if(条件表达式);          //空语句做 if 中的语句

语句;
```

则不管条件表达式为真为假,总是接着执行语句。

3. if…else 语句

if…else 语句的语法为:

```
if(条件表达式)
   //语句 1;
else
   //语句 2;
```

其流程图描述见图 3-2。

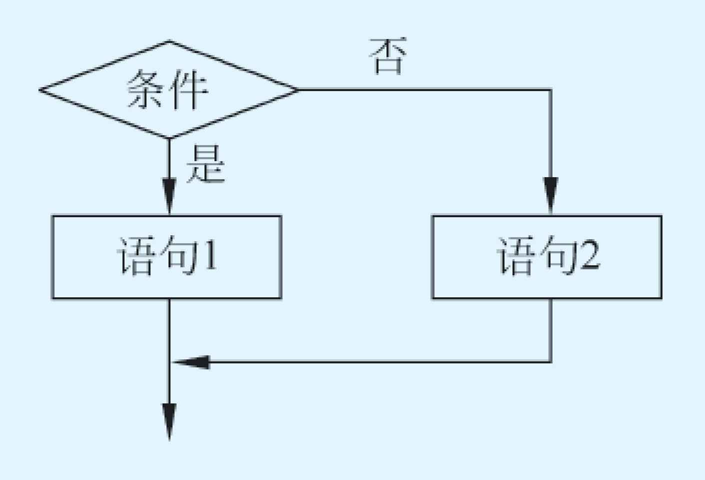
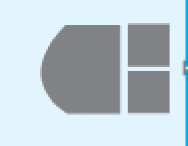


图 3-2 if…else 语句结构

if…else 语句让用户在程序中构造"不是…就是…"的判断点。例如:

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
   cout <<"please input the b key to hear a bell.\n";
   char ch = getche();

   if(ch == 'b') cout <<'\a';
else
   if(ch == '\n')
       cout <<"what a boring select on...\n";
   else
       cout <<"bye!\n";
}
```



该程序等待输入一个字符，如果是'b'，则响铃，否则，如果是回车，则输出"what a boring select on…"，不是，则输出"bye!"。

else 后面的代码规则和 if 后面的代码规则一样。

else 后跟语句。既然是语句，就可以是 if 语句。上例中跟的就是 if 语句。

4. 解决二义性

如果有下面的代码：

```
int x = 20;
if(x >= 0)
   if (x < 50)
      cout <<"x is ok\n";
else
   cout <<"x is not ok\n";
```

编译并不看程序的缩进格式，而只关心语法。该程序能打印什么呢？

该程序有两种解释：

一种是把第 2 个 if 与 else 配对：

```
int x = 20;
if(x >= 0)
{
   if (x < 50)
      cout <<"x is ok\n";
   else
      cout <<"x is not ok\n";
}
```

另一种是把第 2 个 if 包在一个程序块中：

```
int x = 20;
if(x >= 0)
{
   if (x < 50)
      cout <<"x is ok\n";
}
else
      cout <<"x is not ok\n";
```

C++规定，if…else 语句成对的规则是：else 连接到上面第 1 个没有配对的且为可见的 if 上。所以上例的 else 应属于第二个 if 语句，即第一种解释。

又如：

```
if(条件)              //第 1 个 if
   if(条件)           //第 2 个 if
   {
      if(条件)        //第 3 个 if
         语句;
   }
  else
    语句;
```



上例的 else 连到第 2 个 if 上,因为第 3 个 if 不可见。第 2 个 if 是 else 最先碰到的没有配对过的 if。

对于程序的可读性要求来说,无论哪一种情况,都不理想,更好的方法是改变程序的实现。例如,将前面的代码的两个 if 合并,成为下面的代码:

```
int x = 20;
if(x >= 0 && x < 50)
   cout <<"x is ok\n";
else
   cout <<"x is not ok\n";
```

3.7 条件运算符

条件运算符的语法为:

```
(条件表达式)?(条件为真时的表达式):(条件为假时的表达式)
```

例如:

```
x = a < b? a: b;
```

条件运算符构成一个表达式。它是 C++中唯一一个三元运算符,它们之间用"?"和"："隔开。上例中,把 a 和 b 中较小的值赋给 x。该例是 if…else 语句的一个替代:

```
if(a < b)
   x = a;
else
   x = b;
```

条件运算符构成表达式,它是有值的。而 if…else 语句不能有值,所以 if…else 语句不能替代条件运算符。例如,下面的代码不能由 if…else 替代:

```
cout <<(a < b? a:b) << endl;
```

输出语句要打印一个值,该值是 a 与 b 的较小值。由于≪的优先级高于条件运算符,所以输出语句中要将条件运算符构成的表达式用括号括起来。

条件运算符表达式的值与测试值没有直接的关系。例如:

```
cout <<(number == 1?"file":"files") << endl;
```

该输出语句中,条件运算符表达式的条件若成立,取值为"file";否则,取值为"files"。其中,条件为两个整型数的比较,而表达式的值为字符串。

条件运算符可以嵌套。例如:

```
x > y?"great than":x == y?"equal to":"less than"
```

它等价于:

```
(x > y)?"great than":((x == y)?"equal to":"less than")
```

当 x>y 时,值为"great than";x==y 时,值为"equal to";否则,值为"less than"。条


件运算符的嵌套可读性不够好，应通过加括号将意义明确。

在一个条件运算符的表达式中，如果后面两个表达式的值类型相同，均为左值，则该条件运算符表达式的值为左值表达式。例如：

```
int x = 5;
long a,b;
(x?a:b) = 1;            //ok:因为 a 和 b 都是左值
(x?x:a) = 2;            //error:x 和 a 不同类型。编译器将其解释为(long)x 和 a
(x == 2?1:a) = 3;       //error:1 非左值
```

“(x?a:b)=1”表示当 x 为 0 时，b=1，否则 a=1。这里的括号是必需的，否则将被看作“x?a:(b=1)”。“(x?x:a)=2”中，尽管 x 是左值，a 也是左值，但 x 与 a 不同类型，条件运算符要对其进行操作数的隐式转换，使之成为相同的类型。**任何被转换的变量都不是左值**。

→ *在 C 中，条件运算符是不能作左值的，所以“(x?a:b)=1;”将通不过编译。*

3.8 逗号表达式

逗号表达式的语法为：

```
表达式 1, 表达式 2, …,表达式 n
```

C++顺序计算表达式 1，表达式 2，…，表达式 n 的值。例如：

```
int a,b,c;
a = 1, b = a + 2, c = b + 3;
```

由于按顺序求值，所以能够保证 b 一定在 a 赋值之后，c 一定在 b 赋值之后。该逗号表达式可以用下面 3 个有序的赋值语句来表示：

```
a = 1;
b = a + 2;
c = b + 3;
```

逗号表达式是有值的，这一点是语句所不能代替的。逗号表达式的值为第 n 个子表达式的值，即表达式 n 的值。例如：

```
int a,b,c,d;
d = (a = 1,b = a + 2,c = b + 3);
cout << d << endl;
```

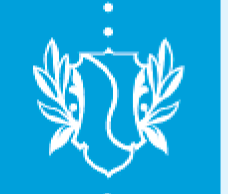
输出结果为：

```
6
```

上例中输出的结果 d 即为 c 的值。

逗号表达式还可以用于函数调用中的参数。例如：

```
func(n,(j = 1,j + 4),k);
```



该函数调用有 3 个参数,中间的参数是一个逗号表达式。括号是必需的,否则,该函数就有 4 个参数了。逗号表达式作为值的形式,可以用于几乎所有的地方。

C++中,如果逗号表达式的最后一个表达式为左值,则该逗号表达式为左值。例如:

```
(a = 1,b,c + 1,d) = 5;    //ok:即 d = 5
```

→ 在 C 中,逗号表达式是不能作左值的,所以"(a=1,b,c+1,d)=5;"将通不过编译。

3.9 求值次序与副作用

在表达式中,各操作数的求值次序并没有在 ANSI C++标准中规定。于是各个编译器为提高产生目标代码的质量,在不破坏操作符的优先级和结合性的前提下,对操作数(是个表达式)访问进行必要的顺序安排。在顺序安排中,操作数要进行挪动操作,可能会经历求值运算,而求值运算如果修改了另一个表达式中的变量,则会产生副作用。

1. 不同的编译器求值顺序不同

例如:

```
int a = 3,b = 5,c;
c = a * b +  ++b;
cout << c << endl;
```

c 是 a * b 和++b 的和。在求和之前,先要把这两个操作数安排在加运算的地方。就在安排的时候,可能先安排前一个表达式 a * b,也可能先安排后一个表达式++b。安排时,要对表达式进行计算求值。可是若先安排后一个表达式,求其值之后,变量 b 内存空间中的值却发生变化。在将 a * b 的值放到参加加运算的位置时,面临求 a * b 的问题。到底 b 是取自最初的 b 变量值,还是在前一个操作数放置到加运算的地方之后(b 已经发生变化),再去取 b 变量的值呢?也就是说,a * b 为 3×5 还是 3×6 呢?见图 3-3。

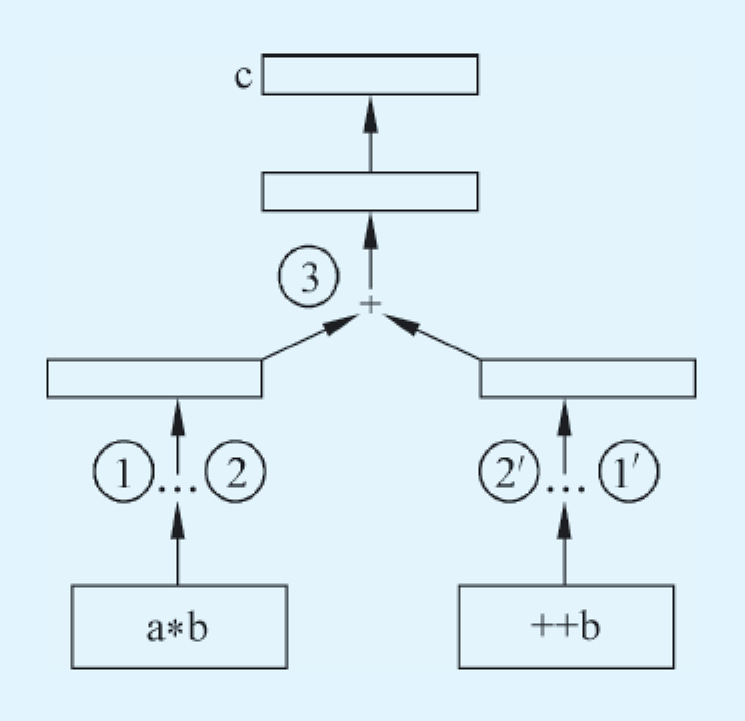


图 3-3　表达式内部的运算次序

编译器可以按①②′③的顺序,也可以按①′②③的顺序来安排求值顺序。

该程序段在 BC 中运行得到 24,而在 VC 中运行却得到 21。



2. 求值顺序使交换律失去作用

加法操作我们都知道 a+b=b+a。也就是说,交换律成立。在 C++ 中,对简单的表达式,交换律是成立的,但对复合表达式,交换律未必成立。例如:

```
c = a * b +  ++b;
```

与:

```
c = ++b + a * b;
```

在 VC 中,运行结果前者为 21,后者为 24。这一现象同样可以用图 3-3 来解释,即先求前操作数的值还是先求后操作数的值,C++ 并无明确规定。

3. 求值顺序使括号失去作用

在表达式中,括号的优先级是最高的。例如,a * (b+c)中,先做加法,后做乘法。在 C++ 中,简单的表达式括号优先可以做到,但复合表达式未必如此。

例如:

```
int a = 3,b = 5,c;
c = ++b * (a + b);
```

预想的操作应为先做 a+b 得 8,然后乘上++b 得 6 * 8=48,但实际在 VC 中却为 54。这并不是说乘法抢先执行(++b * a,然后再加 b),而是括号外面的表达式先行求值,以准备与括号表达式相乘。

4. 消除副作用

在我们举的 3 个例子中,原因主要是++b 引起的。++b 具有变量 b 的修改(副作用)和它所提供的表达式值两个操作。

同样,赋值表达式也会引起副作用。例如:

```
int a,b = 20;
a = (b = 25) + b;
```

a 是等于 25+20,还是 25+25 呢? 分析之后发现,赋值表达式同样有提供表达式值的同时修改变量的行为。

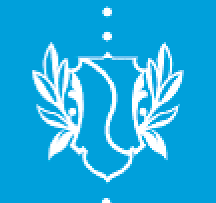
表达式和语句的副作用说明编程者对程序思路还有不够完善、不够周密的地方。它导致可读性下降,也破坏了可移植性。所以编程时务必要避免副作用的产生。

解决表达式副作用的方法是分解表达式语句,即将复合表达式语句写成几个简单的表达式语句。例如,下面的代码用多个语句代替前面有副作用的表达式语句:

```
c = b + a * b; b++ ;
```

或者:

```
b++ ; c = b + a * b;
```



小结

C++为每个运算符规定了一个优先级和结合性,以控制各运算的顺序,确保表达式计算的一致性。利用括号可以改变表达式的运算顺序。

左值是能出现在赋值表达式左边的表达式,它占有内存空间,并且可修改。

如果运算结果超过了该数据类型能够表达的范围,则 C++进行截断处理。

参加运算的两个操作数类型不同时,C++将自动作隐式类型转换,但有时候,不得不作强制类型转换。

前增量操作符通知 C++编译器先增加变量的值,然后再使用变量;后增量操作符通知编译器先使用变量,然后再增加该变量值。

关系运算中,=与==经常要搞错。逻辑运算符 && 和||都是短路运算符。

表达式和语句的一个重要差别是:表达式具有值,而语句是没有值的。

副作用是一个表达式中的嵌套表达式,在提供值的同时,又对某处变量进行修改所引起的。对于副作用,由于其运算结果的不可预料性,所以要尽量避免。

然而,副作用并不是什么都不好,在函数中,正是利用了副作用才使许多代码更精简和可读。事实上函数是产生副作用的温床。指针是最大的"罪魁祸首"。当学习了函数的内部实现机制和指针之后,读者会有所体会。

练习

3.1 写出以下公式的 C++表达式:

(1) $\sqrt{(\sin(x))^{2.5}}$

(2) $\frac{1}{2}\left(ax+\frac{a+x}{4a}\right)$

(3) $\frac{c^{x^2}}{\sqrt{2\pi}}$

在 math.h 头文件中:

正弦函数原型为:double sin(double x)表示 x 弧度的正弦值;

指数函数原型为:double exp(double x)表示 e 的 x 次方;

平方根函数原型为:double sqrt(double x)表示 x 的平方根;

幂指数函数原型为:double pow(double x, double y)表示 x 的 y 次方。

3.2 写出下列表达式的值:

(1)
```
int e = 1,f = 4,g = 2;
float m = 10.5,n = 4.0,k;
k = (e + f)/g + sqrt((double)n) * 1.2/g + m;
```

(2)
```
float x = 2.5,y = 4.7;
int a = 7;
x + a % 3 * (int)(x + y) % 2/4;
```



(3)
```
int a,b;
a = 2,b = 5,a++ ,b++ ,a + b;
```

3.3 写出下列程序的运行结果。

(1)
```
#include < iostream >
using namespace std;
int main()
{
   int a1,a2;
   int i = 5,j = 7,k = 0;
   a1 = !k;
   a2 = i!= j;
   cout <<"a1 = " << a1 <<'\t'
        <<"a2 = " << a2 << endl;
}
```

(2)
```
#include < iostream >
using namespace std;
int main()
{
   int x,y,z;
   x = 1;
   y = 1;
   z = 1;
   x = x||y&&z;
   cout << x <<"," <<(x&&!y||z) << endl;
}
```

(3)
```
#include < iostream >
using namespace std;
int main()
{
   int a,b,c;
   int s,w,t;
   s = w = t = 0;
   a =- 1;
   b = 3;
   c = 3;
   if(c > 0)
   s = a + b;
   if(a < = 0)
   {
   if(b > 0)
   if(c < = 0)
     w = a - b;
   }
   else
    if(c > 0)
       w = a - b;
    else
       t = c;
          cout << s <<','
          << w <<','
          << t << endl;
}
```



(4)
```
#include <iostream>
using namespace std;
int main()
{
    int a,b,c,d,x;
    a = c = 0;
    b = 1;
    d = 20;
    if(a)
    d = d - 10;
    else
       if(!b)
          if(!c)
            x = 15;
          else
            x = 25;
    cout << d << endl;
}
```

3.4　根据以下函数关系,对输入的每个 x 值,求 y 值。请编制此程序。

```
y = x(x + 2)        2 < x ≤10
y = 2x             -1 < x ≤2
y = x - 1           x ≤ -1
```

3.5　编程实现输入一个整数,判断其能否被 3、5、7 整除,并输出以下信息之一:

(1) 能同时被 3、5、7 整除;

(2) 能被其中两数(要指出哪两个)整除;

(3) 能被其中一个数(要指出哪一个)整除;

(4) 不能被 3、5、7 任一个整除。

3.6　编程实现输入一个整数,输出相应的五分制成绩。设 90 分以上为“A”,80 分至 89 分为“B”,70 分至 79 分为“C”,60 分至 69 分为“D”,60 分以下为“E”。

第4章　过程化语句

高级语言源程序的基本组成单位是语句。语句按功能可以分为两类：一类用于描述计算机执行的操作运算(如表达式语句)，即操作运算语句；另一类是控制上述操作运算的执行顺序(如循环控制语句)，即流程控制语句。后一类语句也称为过程化语句。学习本章后，要求掌握 C++各种过程化控制语句结构，并理解常用的过程化程序实例，掌握其开发方法。

4.1　while 语句

while 循环由 4 个部分组成：循环变量赋初值、继续条件、循环体和改变循环变量的值，见图 4-1。

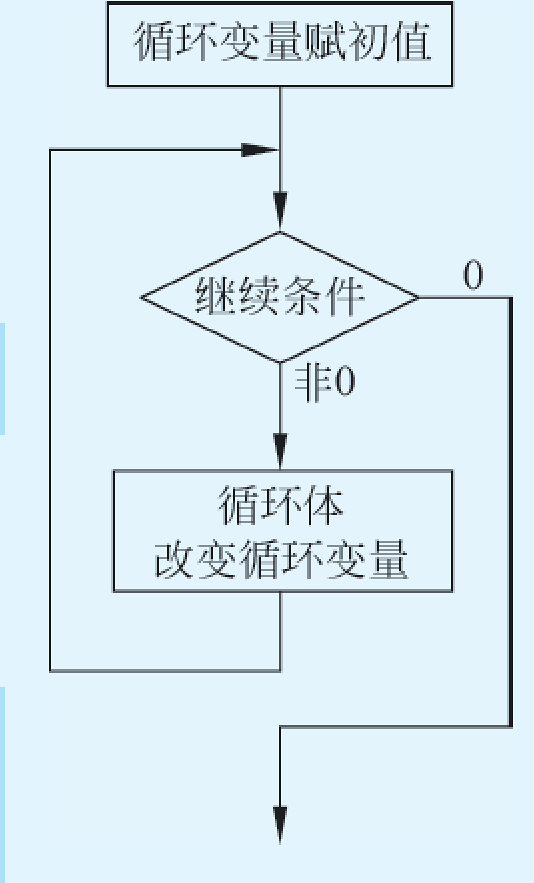


图 4-1　while 循环结构

while 语句的作用是判断一个条件表达式，以便决定是否应当进入和执行循环体，当满足该条件时进入循环，不满足该条件时则不再执行循环。其表现形式为：

```
while(条件表达式)
  循环体
```

语句中的条件表达式就是图中的继续条件。

例如，下面的代码表示了一个 while 循环：

```
i = 1;                      //循环变量赋初值
while(i <= 10)              //继续条件
{                           //循环体
   sum = sum + i;
   i ++ ;                   //改变循环变量的值
}
```

该例是计算 sum＝1＋2＋3＋…＋10 的代码片段。赋初值是对循环变量 i 而言。在开始循环前给控制变量赋初值(i＝1)是重要的，继续条件(i⩽10)决定循环继续多久。通常在继续条件的表达式中，总是包括循环变量。循环体包括在执行循环时将要做的操作。



图 4-1 中的继续条件是一个表达式。当表达式为非 0 时,执行循环体中的语句,否则越过循环。例如,求 1+2+3+…+99+100 的值:

```
//---------------------
//    ch4_1.cpp
//---------------------
#include<iostream>
using namespace std;
//---------------------
int main(){
  int i = 1,sum = 0;     //初始化
  while(i<= 100){
    sum = sum + i;
    i = i + 1;
  }
  cout <<"sum = "<< sum << endl;
}//---------------------
```

运行结果为:

```
sum = 5050
```

如果循环体包含一个以上的语句,应该用大括号括起来,以块语句形式出现。如果不加大括号,则 while 的范围只到 while 后面第一条语句。例如,上例中的循环体可以写成"sum+=i++;"一条语句,所以,循环体可以省略大括号:

```
while(i<= 100)
  sum += i++;
cout <<"sum = "<< sum << endl;
```

循环体中应该有使循环趋向结束的语句。上例中,i 的初值为 1,循环结束的条件为不满足 i≤100,随着每次循环都改变 i 的值,使得 i 的值越来越大,直到 i>100 为止。如果没有循环体中的"i=i+1;",则 i 的值始终不改变,循环就永不终止。

→ 不必要的优化

while 循环中的继续条件是一个表达式,并没有更多的限定,所以,上例可以在继续条件处放上一个逗号表达式以完成同样的功能:

```
i = 1;
while(sum += i++,i<= 100);

cout <<"sum = " << sum << endl;
```

该代码的 while 语句中的循环体为一个分号,代表空语句。

根据逗号表达式的概念,C++将顺序执行逗号表达式中每个子表达式,并以最后一个子表达式的值作为整个逗号表达式的值。因此,继续条件中的逗号表达式的值是 i≤100,它在前一个子表达式 sum+=i++执行后求取,i 在此处无副作用。

C++有足够的能耐让代码最大限度的优化。该代码也显示了 C++的灵活与技巧,但可读性较差,所以它不是现代程序设计所追求的。我们介绍的用意是让初学者见识这类代码,以达到更好地领会概念的目的。

4.2 do…while 语句

do…while 语句的表现形式为：

```
do
   循环体
while(条件表达式)
```

当流程到达 do 后，立即执行循环体语句，然后再对条件表达式进行测试。若条件表达式的值为真(非 0)，则重复循环，否则退出。

该语句结构使循环至少执行一次。

例如，要从键盘中得到一个范围为 1～10 的数：

```
//-----------------------------------
//                 ch4_2.cpp
//-----------------------------------
#include <iostream>
using namespace std;
//-----------------------------------
int main(){
   int val;
   do {//循环体
      cout <<"please enter a number between 1 and 10\n";
      cin >> val;               //修改条件
      if(val < 1 || val > 10)
         cout <<"the number be not between 1 and 10\n";
   }while( val < 1 || val > 10 ); //继续条件
   cout <<"you entered a " << val << endl;
}//-----------------------------------
```

运行结果为：

```
please enter a number between 1 and 10
12
the number be not between 1 and 10
please enter a number between 1 and 10
6
you entered a 6
```

该程序读入一个 1～10 的数，满足条件后就越过循环，执行显示读入的数值。do…while 循环结构见图 4-2。

do…while 循环至少执行一次，因为直到程序到达循环体的尾部遇到 while 时，才知道继续条件是什么。如果继续条件仍然成立，程序回转到 do…while 循环的顶部，继续循环。



在循环体中,if 语句的条件和 while 的继续条件是同一个,那只是一个巧合,并非必须。

代码中,继续条件的不断变化很重要。如果 val 值恒定不变,则继续条件也永不改变,导致死循环。

do…while 循环同样需要循环变量赋初值。do…while 循环在循环体的底部进行继续条件的测试,所以它至少将执行一次循环体。而 while 语句在循环的顶部测试,有可能永远不执行循环体。

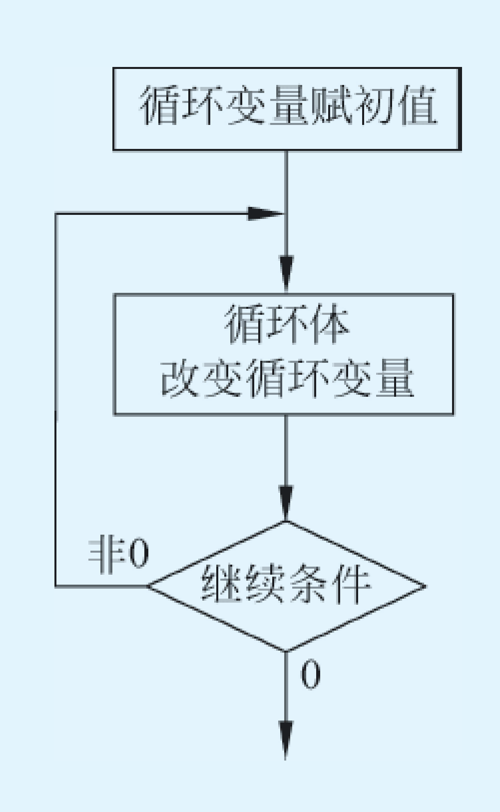


图 4-2　do…while 循环结构

while 在许多场合可以做 do…while 能做的事,反之亦然。例如,4.1 节中求 sum= $\sum_{n=1}^{100} n$。

```
//----------------------------------
//              ch4_3.cpp
//----------------------------------
#include <iostream>
using namespace std;
//----------------------------------
int main(){
   int i = 1, sum = 0;
   do {
      sum = sum + i;
      i = i + 1;
   }while(i <= 100);
   cout <<"sum = " << sum << endl;
}//----------------------------------
```

运行结果为:

```
sum = 5050
```

→ 书写格式与可读性

do…while 循环中,while(继续条件)后面的分号不要遗忘。不要把 do…while 循环与 while 循环使用空语句作为循环体的形式相混淆。

```
do sum += i++;
while(i <= 100);             //do…while 求和代码段

while(i++ < 10000);          //while 时间延迟代码段
```

因为它们从局部看都是:

```
while(表达式);
```

为明显区分它们,do…while 循环体即使是一个单语句,习惯上也使用大括号包围起来,并且 while(表达式)直接写在大括号"}"的后面。这样的书写格式可以与 while 循环清

楚地区分开来。例如：

```
do
{
   sum += i ++ ;
}while(i <= 100);
```

4.3 for 语句

可以用图来描述 for 循环结构，见图 4-3。

for 语句的一般表现形式为：

```
for(表达式 1; 表达式 2; 表达式 3)
   循环体
```

它的执行过程如下：

(1) 求解表达式 1；

(2) 求解表达式 2，若为 0，则结束循环，转到(5)；

(3) 若表达式 2 为真，执行循环体，然后求解表达式 3；

(4) 转回(2)；

(5) 执行 for 语句下面的一个语句。

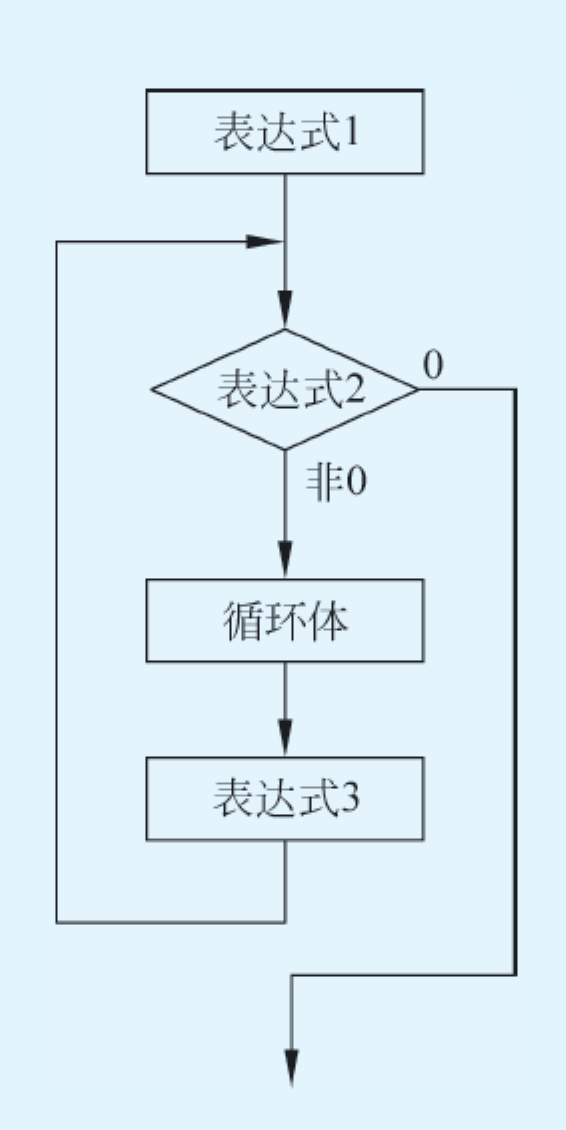


图 4-3 for 循环结构

C/C++中的 for 语句相对 while 和 do…while 来说，较为灵活。它不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。

例如，for 循环对于求和来说，方式更简单、可读：

```
for(i = 1; i <= 100; i ++ )  //初始化,继续条件,步长都在顶部描述
{
   sum += i;                  //循环体相对简洁
}
```

如果将 for 语句的一般形式用 while 来表达，则为如下：

```
表达式 1;
while(表达式 2)
{
   循环体
   表达式 3;
}
```

所以 for 语句是将循环体所用的控制放在循环顶部统一表示，显得更直观。除此之外，for 语句还充分表现了其灵活性：

(1) 表达式 1 可以省略。此时应在 for 语句之前给循环变量赋初值。若省略表达式 1，其后的分号不能省略。

例如，求和运算：

```
i = 1;
for( ; i <= 100; i ++ )       //分号不能省略
   sum += i;
```



执行时,跳过求解表达式 1 这一步,其他不变。由于循环体由一条语句构成,所以大括号可以省略。

(2) 表达式 2 可以省略。即不判断继续条件,循环无终止进行下去。也就是认为表达式 2 始终为真。这时候,需要在循环体中有跳出循环的控制语句。

例如,求和运算:

```
for(i=1; ; i++)          //分号不能省略
{
   sum+=i;
   if(i>100)
       break;
}
```

等价于:

```
for(i=1; 1; i++)         //表达式 2 为真(1)
{
   sum+=i;
   if(i>100)
       break;
}
```

此处 break 表示退出循环,见第 4.5 节。

(3) 表达式 3 可以省略。但此时程序员应另外设法让循环变量递进变化,以保证循环能正常结束。

例如,求和运算:

```
for(i=1; i<=100; )    //分号不能省略
   sum+=i++;          //同时改变循环变量
```

在循环体中,必须自己对循环变量进行修改(i++),其效果与在表达式 3 上设置是一样的。

(4) 表达式 1 和表达式 3 可同时省略。

例如,下面的代码同样能完成求和运算:

```
for( ; i<=100; )
   sum+=i++;
```

(5) 3 个表达式都可省略。即不设初值,不判断条件(认为表达式 2 为真),循环变量不增值,无终止执行循环体。

例如,求和运算也可以这样:

```
for( ; ; )
{
   sum+=i++;
   if(i>100)
     break;
}
```

(6) 表达式 1、表达式 2、表达式 3 都可以为任何表达式。

例如,求和运算中可设置 sum 的初值:

```
for(sum = 0; i <= 100; i++)
    sum += i;
```

例如，表达式 1 为逗号表达式：

```
for(sum = 0, i = 1; i <= 100; i++)
    sum += i;
```

例如，表达式 1 和表达式 3 都为逗号表达式：

```
for(i = 0, j = 100, k = 0; i <= j; i++, j--)
    k += i * j;
```

例如，表达式 2 和表达式 3 可以为赋值或算术表达式的情况，下面两个语句都可以完成同样的求和运算：

```
for(i = 1; i <= 100; sum += i++);      //循环体为空语句

for(i = 1; sum += i++, i <= 100; );    //表达式 3 省略，循环体为空语句
```

注意，在了解以上各种编程方法的同时，不要忘了可读性。

(7) 表达式 1 可以是循环变量定义。C++的变量定义可以在任何语句的位置，for 循环中也不例外。例如，下面的代码完成求和运算：

```
for(int i = 1; i <= 100; i++)
    sum += i;
```

for 循环使得所有的循环控制细节都可在语句中描述，程序又精炼又可读。只要循环变量不在程序的其他地方使用，在 for 头部定义循环变量是最好的，该变量只在循环体中有效，循环退出后自行消失。关于作用域规则见第 6.3 节。

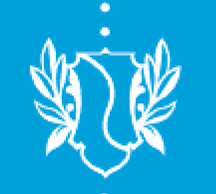
4.4 switch 语句

switch 语句是多分支选择语句。if 语句是二分支选择语句，但在实际问题中常常需要用到多分支的选择。例如，学生成绩分类(85 分以上为 A，70 分至 84 分为 B，60 分至 69 分为 C 等)，人口统计分类(按年龄分为老、中、青、少、儿、幼)等。嵌套的 if 语句也可以处理多分支选择，但是，switch 更加直观。

switch 语句的一般形式为：

```
switch(表达式)
{
    case 常量表达式 1: 语句组 1
    case 常量表达式 2: 语句组 2
    ...
    case 常量表达式 n: 语句组 n
    default: 语句组 n+1
}
```

例如，根据考试成绩的等级输出百分制分数段：




```
char grade;
//...

switch(grade)
{
   case'A': cout <<"85～100\n";
   case'B': cout <<"70～84\n";
   case'C': cout <<"60～69\n";
   case'D': cout <<"< 60\n";
   default: cout <<"error\n";
}
```

(1) switch 后面括号中的表达式只能是整型、字符型或枚举型。case 后面的常量表达式类型必须与其匹配。例如,下面的代码错误地用浮点类型作 switch 的表达式,它会引起编译错误:

```
float f = 4.0;

switch(f) //error
{
   //...
}
```

(2) 当表达式的值与某一个 case 后面的常量表达式值相等时,就执行此 case 后面的语句,若所有 case 中的常量表达式值都没有与表达式值匹配,就执行 default 后面的语句。

(3) **case 语句起标号的作用**。标号不能重名,所以每一个 case 常量表达式的值必须互不相同,否则就会出现编译错误。例如,下面的代码中 case 出现相同常量值:

```
case 'A': cout <<"this is A\n";
case 65 : cout <<"this is 65\n";     //error:'A'等值于 65
```

(4) 因为 case 语句起语句标号的作用,所以 case 与 default 并不改变控制流程。

例如,在最初的例子中,若 grade 的值等于'A',则将连续输出:

```
85～100
70～84
60～69
< 60
error
```

case 通常与 break 语句联用,以保证多路分支的正确实现。例如,改写上例以使输出某个成绩段后终止 switch 语句:

```
char grade = 'B';

switch(grade)
{
   case'A': cout <<"85～100\n"; break;
   case'B': cout <<"70～84\n"; break;
   case'C': cout <<"60～69\n"; break;
   case'D': cout <<"< 60\n"; break;
   default: cout <<"error\n";
}
```



输出结果为：

```
70～84
```

最后一个分支可省略 break 语句。

（5）各个 case（包括 default）的出现次序可以任意。在每个 case 分支都带有 break 的情况下，case 次序不影响执行结果。

例如，上面的代码可以写成下面的代码：

```
char grade;
//...

switch(grade)
{
   case'C': cout <<"60～69\n"; break;
   default: cout <<"error\n"; break;
   case'D': cout <<"< 60\n"; break;
   case'A': cout <<"85～100\n"; break;
   case'B': cout <<"70～84\n"; break;
}
```

当 grade 为'B'值时，输出结果也为：

```
70～84
```

（6）多个 case 可以共用一组执行语句。例如：

```
//...
case'A':
case'B':
case'C': cout <<"> 60\n";
```

当 grade 的值为'A'、'B'和'C'时，都输出“> 60”。

几个状态都执行同一操作时，不能将值用逗号隔开，图谋在一个 case 中实现。例如：

```
case 1,2,3: cout <<"hello";   //error
```

是错的，应为：

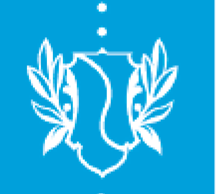
```
case 1:
case 2:
case 3: cout <<"hello";
```

（7）default 语句是可选的。当 default 不出现时，则当表达式的值与所有常量表达式的值都不相等时，越过 switch 语句。

（8）switch 不一定非要包含复合语句块。例如，下面两条语句是等价的：

```
switch(i) case 1: cout <<"ok\n";
if(i == 1) cout <<"ok\n";   //与上一条语句等价
```

（9）switch 语句可以嵌套。case 与 default 标号是与包含它的最小的 switch 相联系的。例如：




```
int i,j;
//...

switch(i)
{
   case 1:  //...
   case 2:
     switch(j)    //嵌套 switch
     {
       case 1:  //...
       case 2:  //...
       //...
     }
     case 3:  //...
   //...
}
```

switch(j)中的 case 1 标号不会与外面的 switch(i)中的 case 1 标号相混淆。

(10) 用 if 语句与 switch 语句可以互相补充。

例如,根据学生的分数输出其成绩等级:

```
int grade;
//...

if(grade>=85 && grade<=100)
   cout<<"A\n";
else if(grade>=70 && grade<85)
   cout<<"B\n";
else if(grade>=60 && grade<70)
   cout<<"C\n";
else if(grade<60 && grade>=0)
   cout<<'D';
else
  cout<<"error\n";
```

最后的 else 等价于 switch 中的 default。

由于 grade 是表示分数,若用 switch 中的 case 标号来表达,则标号需要很多,程序会很长。因为 switch 语句只能对等式进行测试,如果测试值包含一个较大的范围,就需要关系表达式比较,这时候用 if 语句较好。

if…else 语句的执行体等价于 switch 语句的 case 中含有 break 的语句组。

写成上面这样的格式,可以提高 if 语句的可读性。

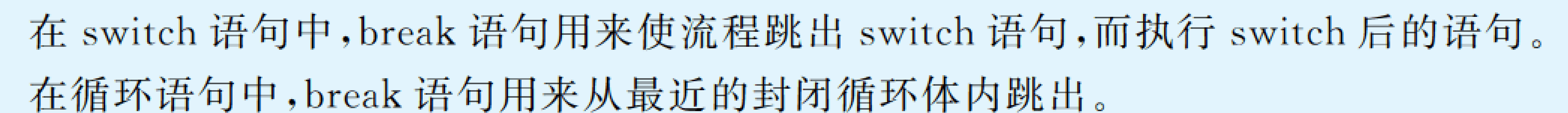
另外,switch 语句只能对整型数进行测试,如果对浮点数进行测试,也需要 if 语句。

若测试一个整型变量取几个不同的值,用 switch 语句比较简明。

4.5 转向语句

1. break 语句

break 语句用在 while、do…while、for 和 switch 语句中。



在 switch 语句中，break 语句用来使流程跳出 switch 语句，而执行 switch 后的语句。

在循环语句中，break 语句用来从最近的封闭循环体内跳出。

例如，下面的代码在执行了 break 语句之后，继续执行“a=1;”处的语句，而不是跳出所有的循环：

```
for( ; ; )
{
   for ( ; ; )
   {
       //...
       if(i==1)
        break;
       //...
   }
   a=1;      //break 跳至此处
   //...
}
```

2. continue 语句

continue 语句用在循环语句中，作用为结束本次循环，即跳过循环体中尚未执行的语句，接着进行下一次是否执行循环的判定。

例如，下面的代码把 100～200 中不能被 3 整除的数输出：

```
for(int n=100; n<=200; n++)
{
   if(n%3==0)
         continue;

   cout << n << endl;
   //...
}
```

当 n 被 3 整除时，执行 continue 语句，结束本次循环，即跳过 cout 语句。只有 n 不能被 3 整除时，才执行 cout 函数。

由于多条语句可以组成块，所以上述代码也可以写成：

```
for(int n=100; n<=200; n++)
{
   if(n%3!=0)        //条件相反
   {
         cout << n << endl;
         //...
   }
}
```

从而省略了 continue 语句。事实上，由于在 C++中有块语句的支持，所以经常使用反条件的 if 语句，把 continue 后面的语句以块的形式包含在 if 语句之中，可以避免使用 continue 语句。

continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不是终止整个循环的执行；而 break 语句则是终止整个循环，不再进行条件判断。对于 for 语句，其二者

的差别见图 4-4。

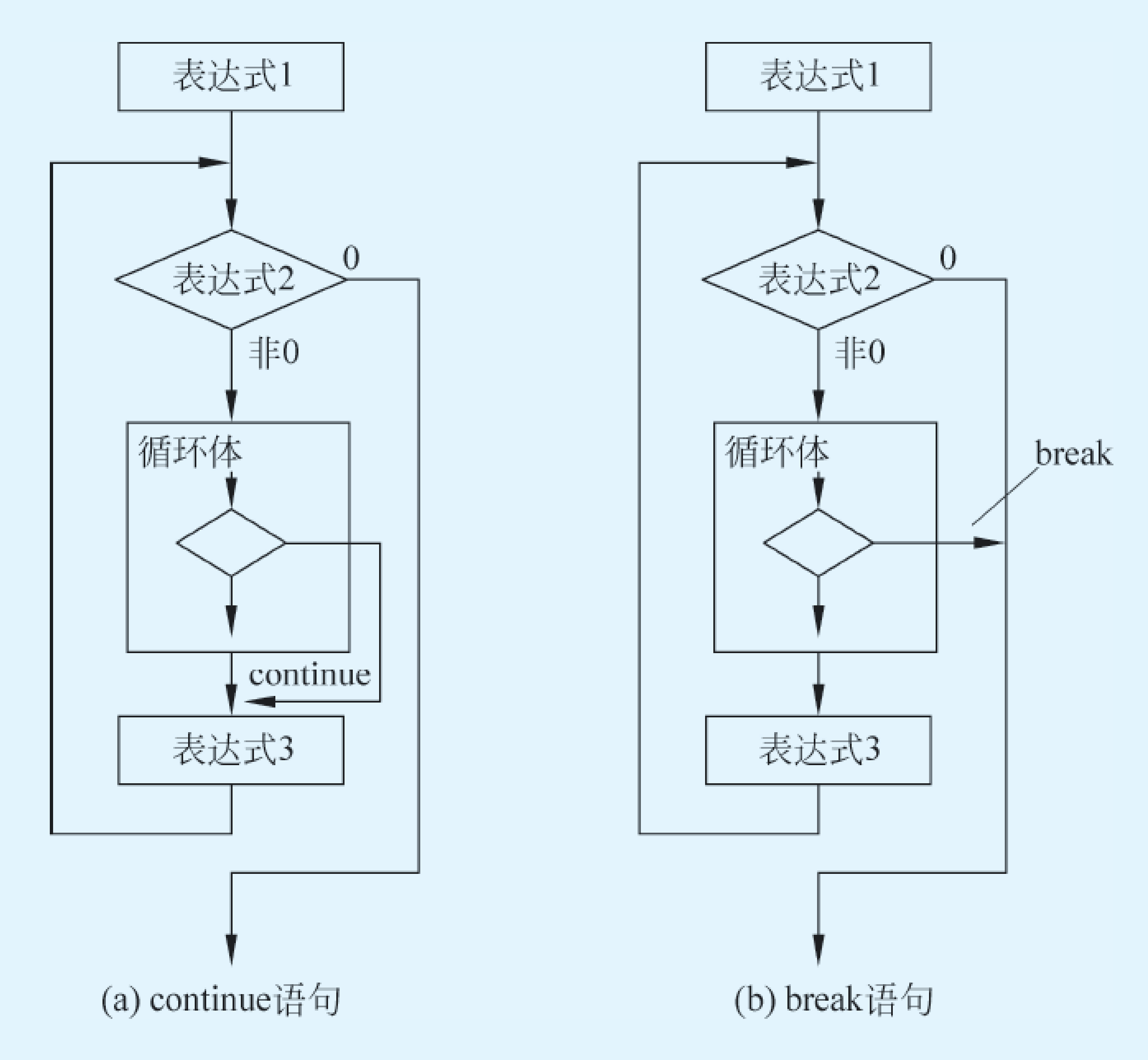


图 4-4 continue 与 break 语句的区别

3. goto 语句

goto 语句将控制从它所在的地方转移到语句标号处。例如,求 1 加到 100 的和:

```
   i = 1; sum = 0;

Loop:                 //语句标号
   sum += i++;
   if(i <= 100)
     goto Loop;
   cout <<"sum is" << sum << endl;
```

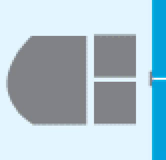
语句标号用标识符表示,它的命名规则与变量名相同。

用 goto 语句实现的循环完全可以用 while 或 for 循环来表示。现代程序设计方法主张限制使用 goto 语句,因为滥用 goto 语句将使程序流程无规则,可读性差。goto 语句只在一个地方有使用价值:当要从多重循环深处直接跳转到循环之外时,如果用 break 语句,将要用多次,而且可读性并不好,这时 goto 可以发挥作用。

例如,下面的代码找到满足乘积为 27 的两个小于 10 的整数后即输出该数,运行结束:

```
for(int i = 1; i < 10; i++)
   for(int j = 1; j < 10; j++)
       if(i * j == 27)
         goto End;

End:        //循环体外
   cout << i <<" * " << j <<"27\n";
```



4.6 过程应用：求 π

程序设计的目的是用正确的方法解决实际问题。之所以强调正确，是因为不同的程序设计方法，难易相差很大，程序复杂性也相差很大。

一个复杂性不大的小问题，可以看作是一个过程，该过程具有输入、处理和输出。

对于问题的求解，输入对应问题给出的条件，处理对应求解问题的算法，输出对应问题的解。

对于程序的描述，输入对应数据定义和初始化，处理对应结构语句的一个序列，输出对应打印输出语句。

对于软件工程来说，程序设计只是其中的一个环节。程序设计的任务是根据给定的数据定义和算法(程序模块)，实现程序的编码和调试。我们学习的程序设计，所指的概念包括了程序设计方法。

程序设计方法有 3 个层次：

(1) 简单的问题求解分析方法(过程化方法)。它适用于简单、孤立的问题求解。一般定义 2～3 个函数便可解决。

(2) 结构化程序设计方法。它适用于一个问题大小适中，能够方便地分解成相对独立的几个功能模块，从而用几个程序文件分别描述并调试实现之。

(3) 面向对象程序设计方法。它面向求解一个用常规方法并不能简单理清头绪的问题。它将问题看作包含若干个小对象的大对象，层层分解对象，研究里面的数据和行为。当一个问题分解成不同层次的对象结构时，程序设计的描述也同时完成。

这里介绍的是第 1 种方法。

例如，用公式$\frac{\pi}{4}\approx 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$求 π 的近似值，直到最后一项的绝对值小于 10^{-8} 为止。

分析：

(1) π 的表示用 double 型。因为 float 型的有效位数是 7 位，而该问题中的最小项的精度要求达到小数点后 8 位。

(2) 先求 π/4，然后求 π。

(3) 分析数列的通项：数列的第 1 项是 1，第 2 项是 −1/3，…，第 n 项是$(-1)^{n-1}/(2\times n-1)$。第 n 项与第 n−1 项的关系为符号变一下，分母加 2。

根据前后项的关系，可以设计一个循环，每次循环将原项分母加 2，符号变更一下，求得新项。最初的分母变量(类型为 long)值是 1。最初的符号变量值是 +1。于是可以用图 4-5 所示的框图来表示算法。

根据该算法，添上头文件，定义相应的变量，实现 while 循环，最后根据 π/4，输出 π 值。

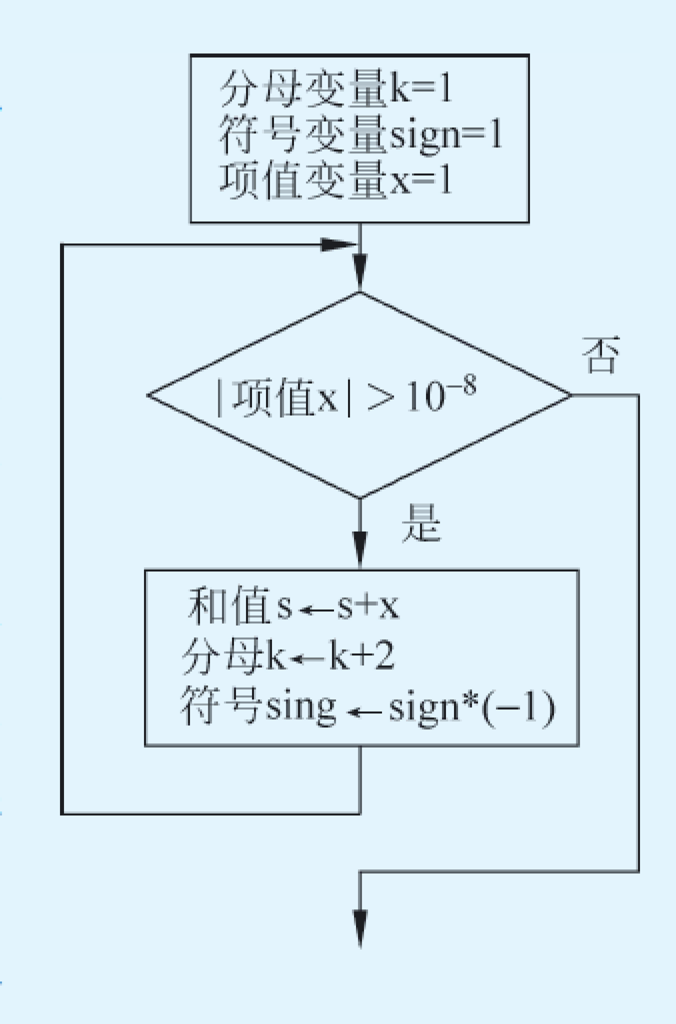


图 4-5 求 π/4 算法一

```
//------------------------------------
//              ch4_4.cpp
//------------------------------------
#include <iostream>
#include <iomanip>                    //使用 setprecision()
#include <cmath>
using namespace std;
//------------------------------------
int main()
{
   double s=0,x=1;                    //初始值
   long k=1;
   int sign=1;

   while(abs(x)>1e-8)                 //项值在比较前要先求绝对值
   {
         s+=x;
         k+=2;
         sign*=-1;
         x=sign/double(k);            //强制转换使 x 得到浮点数值
   }

   s*=4;                              //π 值
   cout<<"the pi is "<<s<<endl;       //输出
       <<fixed<<setpreasion(8)<<s<<endl;
  }//------------------------------------
```

运行结果为：

```
the pi is 3.14159263
```

该程序中的变量名意义在前面的框图中描述。如果只有源程序，那么变量名的命名要能反映出其意义，以使程序能被人读懂。

求解的算法一般都不是唯一的。如果注意到前后项关系的另一种表达：设第 n－1 项为 x，则下一项为 x＊(－1)＊(2＊n－3)/(2＊n－1)，则 for 循环的结构亦很自然，见图 4-6 描述。

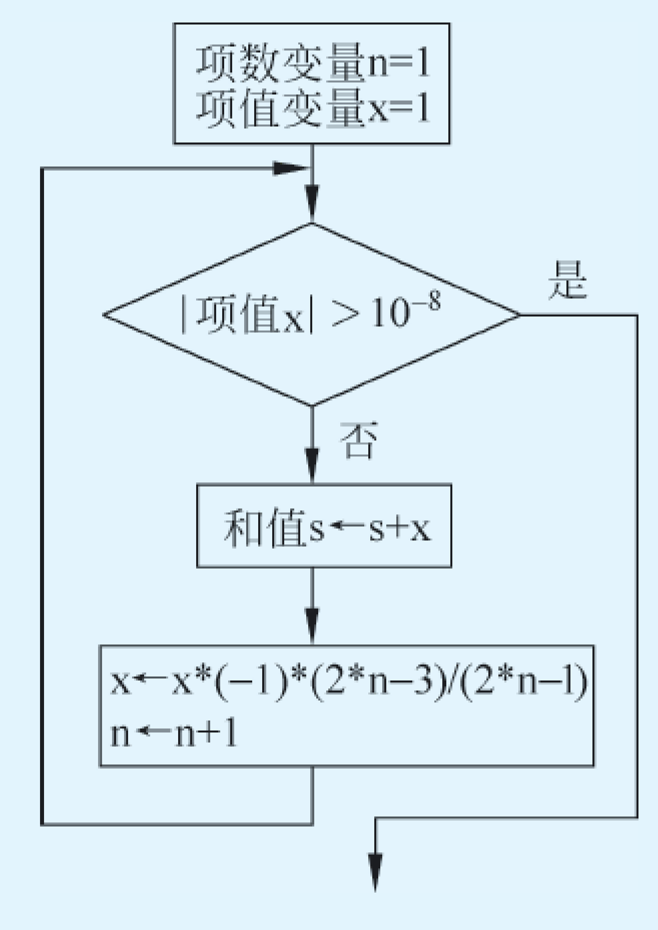


图 4-6　求 π/4 算法二

根据此算法,可以得到下面的程序:

```
//------------------------------------
//              ch4_5.cpp
//------------------------------------
#include<iostream>
#include<cmath>                     //用到abs()
using namespace std;
//------------------------------------
int main(){
   double s=0,x=1;                  //初始值
   for(int n=1; fabs(x)>1e-8; n++,x*=(-1.0)*(2*n-3)/(2*n-1))
        s+=x;

   s*=4;                            //π值
   cout<<"the pi is "<<s<<endl;     //输出
}//------------------------------------
```

运行结果为:

```
the pi is 3.14159
```

如果从效率上去分析,该程序不如前者,因为求项数时要做4次乘法、1次除法,而ch4_4.cpp中只做1次除法。

在步长表达式中,—1.0表示一个浮点数,以使项值x能取到浮点数值。

for的循环体只有一条语句,所以就省去了大括号。

该程序与上面的程序比较,可读性稍好。在算法分析时,多考虑效率,但编程中,我们要更多考虑可读性。从这个意义上来说,本程序比上个程序优。然而,本程序还不是最优的,它完全可以做到既不损失效率,也不损害可读性。

如果条件指明到n=1000000项停止,则程序如何修改,哪个程序更方便修改?请读者思考。

4.7 过程应用:判明素数

给定一个整数m,判断其是否为素数。

分析:m是素数的条件是不能被2、3、…、m—1整除。

根据这一条件,可以立即写出一个循环判明该数是否为素数:

```
//------------------------------------
//              ch4_6.cpp
//------------------------------------
#include<iostream>
using namespace std;
//------------------------------------
int main(){
   //输入
   long m,i;
```




```
    cout <<"please input a number:\n";
    cin >> m;

    //处理
    for(i = 2; i < m; i ++ )          //找 m 的因数
        if(m % i == 0)
            break;

    //输出
    if(m == i)                        //判断 m 是否被小于 m 的数整除
        cout <<"m is prime.\n";
    else
      cout <<"m isn't prime.\n";
}//-------------------------------
```

在程序的输出部分，判断是否 m==i，for 循环有两种退出的情况：一种是不满足 i<m 的循环条件而正常退出，此时 i 正好等于 m；另一种是发现 m 整除 i 时的退出，此时 i 小于 m。所以，判断 m 与 i 是否相等，就能知道 m 是否为素数。

该程序最直接反映了数学定义。但是，当给定的 m 很大时，运算量也很大，能否改进一下算法，使运算量急剧下降？

假定 m 不是素数，则可表示为 $m=i*j$，$i\leqslant j$，则 $i\leqslant\sqrt{m}$，$j\geqslant\sqrt{m}$。

也就是说，如果 m 不是素数，一定能在$\sqrt{m}$内找到一个整数 i 能整除 m，即 m%i 为 0。

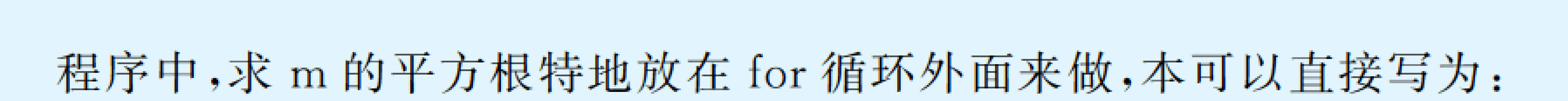
于是，循环可以在 $2\sim\sqrt{m}$ 内进行。

改进的算法可以写为：

```
//-------------------------------
//               ch4_7.cpp
//-------------------------------
#include <iostream>
#include <cmath>                         //用到 sqrt()
using namespace std;
//-------------------------------
int main(){
    //输入
    long m;
    cout <<"please input a number:\n";
    cin >> m;

    //处理
    double sqrtm = sqrt(m);              //用到 math.h
    for(int i = 2; i <= sqrtm; i ++ )
        if(m % i == 0)
            break;

    //输出
    if(sqrtm < i)                        //注意 == 与 =
        cout <<"m is prime.\n";
    else
        cout <<"m isn't prime.\n";
}//-------------------------------
```



程序中,求 m 的平方根特地放在 for 循环外面来做,本可以直接写为:

```
for(int i = 2; i < sqrt(m); i ++ )
```

但这样每次比较都要求一次平方根。为了明显提高循环的效率,在可读性不受影响的前提下,可以适当对程序进行优化。

如果要求 a～b 的数段内所有的素数,则应该对每个 a～b 中的数(循环)都进行上述程序的判断(循环),所以它为二重循环。

事实上,a～b 的循环步长可以是 2,因为偶数不是素数。但循环前,先要判明 a 是否为偶数。程序如下:

```
//------------------------------------
//              ch4_8.cpp
//------------------------------------
#include <iostream>
#include <iomanip>              //用到 sqrt(double)
#include <cmath>
using namespace std;
//------------------------------------
int main(){
   //输入
   long a,b,l = 0;
   cout <<"please input two numbers:\n";
   cin >> a >> b;
   cout <<"primes from " << a <<" to " << b <<" is:\n";

   //处理
      if(a == 2)
        cont <<"2";
       if(a % 2 == 0)           //单判唯一的一个偶数素数,是则增 1
         a ++ ;

   for(long m = a; m <= b; m += 2)  //步长为 2
   {
     int sqrtm = sqrt(m);
     int i;
     for(i = 2; i <= sqrtm; i ++ ) //判明素数
       if(m % i == 0)
         break;

     //输出
     if(i > sqrtm)              //素数
     {
       if(l ++ % 10 == 0)
         cout << endl;
       cout << setw(5) << m;
     }
   }
}//------------------------------------
```



运行结果为：

```
c:> ch4_8
please input two numbers:
12 78
primes from 12 to 78 is:
    13   17   19   23   29   31   37   41   43   47
    53   59   61   67   71   73
```

该程序中设置了一个打印位置变量，每次打印一个素数，该变量加1，当该变量满足模10为0时，打印一个回车。该代码起到控制每行输出固定数据个数的作用。

程序要运行可靠，输入的校验很重要，如果本程序输入87和12，则该程序会有什么反应，能否改进程序使之对输入进行校验？

4.8 过程应用：求积分

数学上求积分靠公式推导，求一条函数曲线f(x)在x轴上a～b的投影所包围之面积可以看成是一重积分。我们用一种变步长的辛普生递推公式来求解积分问题。积分问题为将该积分看作求如图4-7所示的封闭区域面积。

求解积分的步骤如下：

(1) 用梯形公式计算面积近似值。

$$I_n = T_n = \frac{h}{2}(f(a) + f(b))$$

其中 $n=1, h=b-a$，见图4-8所示。

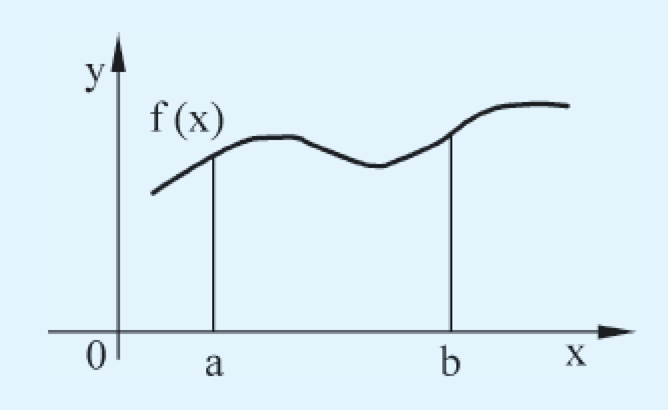


图4-7 [a,b]区域的f(x)函数所包围的面积

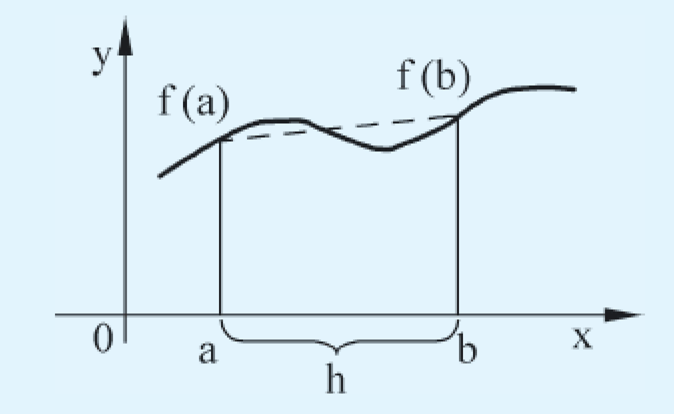


图4-8 用梯形公式计算面积

(2) 用变步长梯形法计算面积近似值。

$$T_{2n} = \frac{T_n}{2} + \frac{h}{2}\sum_{k=0}^{n-1} f\left(x_k + \frac{h}{2}\right)$$

2n意味着将区间[a,b]划分成2n等分。显然开始时，$n=1$，即 $2n=2$ 等分。该公式是说，一半的面积由前面的近似公式给出，另一半则由原n等分的中值小矩形和的一半给出，见图4-9。

图中，$x_0=a, x_n=b, h=(b-a)/n$。中值小矩形即某等分中线的函数值与h的乘积。

(3) 用辛普生公式计算积分近似值。

$$I_{2n} = \frac{4T_{2n} - T_n}{3}$$

(4) 只要上次求的积分值与本次求的积分值之差在一个非常小的ε范围内，则认为所

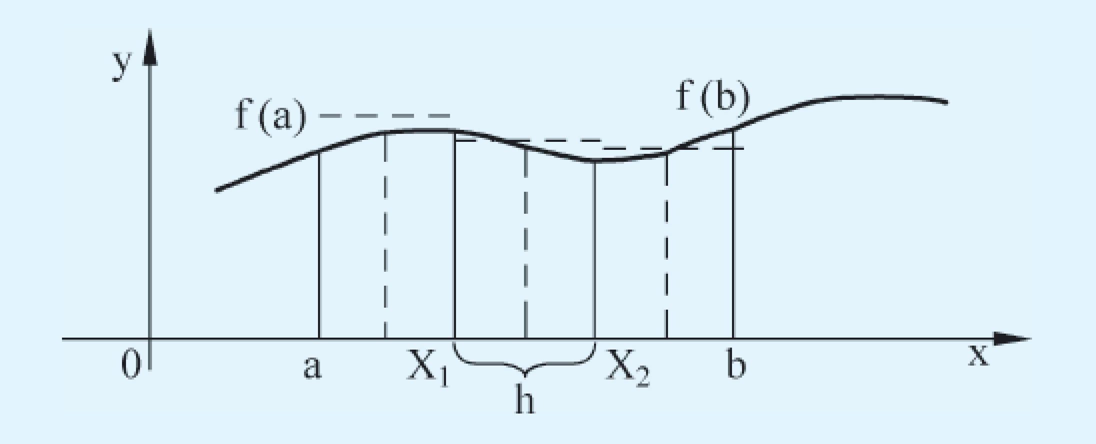


图 4-9　变步长梯形法计算区域划分

求积分的近似度已经达到要求。即若

$$|I_{2n} - I_n| < \varepsilon$$

则结束，I_{2n}即为所求。否则，$n \leftarrow 2n$，$h \leftarrow h/2$，重复第(2)步和第(3)步。

下面的程序是求积分

$$I = \int_0^1 \frac{e^x}{1+x^2} dx$$

其中 ε 取 10^{-8}。

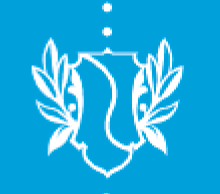
```
//------------------------------------
//                ch4_9.cpp
//------------------------------------
#include <iostream>
#include <iomanip>            //用到 setw()和 setprecision()
#include <cmath>              //用到 abs(double)
using namespace std;
//------------------------------------
double f(double x){return exp(x)/(1 + x * x);}
const double eps = 1e - 8;        //常量 eps 描述计算精度
//------------------------------------
int main(){
   int n = 1;                     //初值
   double a = 0, b = 1;
   double h,Tn,T2n,In,I2n;

   const double eps = 1e - 8;

   h = b - a;
   T2n = I2n = h * (f(a) + f(b))/2;
   In = 0;

   while(abs(I2n - In)> = eps)       //求积分
   {
       Tn = T2n;
       In = I2n;

       double sigma = 0.0;
       for(int k = 0; k < n; k ++ )   //求变步长梯形的和部分
       {
         double x = a + (k + 0.5) * h;
```




```
            sigma += f(x);
        }
        T2n = (Tn + h * sigma)/2.0;               //变步长梯形
        I2n = (4 * T2n - Tn)/3.0;                 //辛普生公式

        n *= 2;                                   //划分
        h/ = 2;
    }

    cout <<"the integral of f(x) from "<< a <<" to " << b <<" is \n"
         << setiosflags(ios::fixed)<< setprecision(8)<< setw(10) << I2n << endl;   //输出结果
}//-----------------------------------

double f(double x)
{
    return exp(x)/(1 + x * x);
}
```

运行结果为：

```
the integral of f(x) from 0 to 1 is
1.27072414
```

该程序将所求积分的函数 f(x)从程序的主函数中分离出来，以便当所求积分的函数发生变更时，只要 f(x)函数的定义更改即可。

在求积分的循环中，初始值 T2n 和 I2n 首先被赋给 Tn 和 In，以实现新的一轮逼近积分值的循环。考虑到这一点，所以在 while 循环之前，先将初始的梯形公式值赋给 T2n 和 I2n。

该循环可以用 for 循环来实现：

```
for(T2n = I2n = h * (f(a) + f(b))/2; fabs(I2n - In)> = eps; n *  = 2, h/ = 2)
{
    //...
}
```

小结

循环是一组语句，计算机反复执行这组语句直到满足终止条件为止。

可以通过循环变量来控制循环。如果事先不知道循环次数，可以在循环体中通过条件判断中间跳转的方法终止循环。

while、do⋯while 和 for 语句都是循环语句，它们可以相互替代，选择其中之一来实现的原因，往往出于代码风格、描述习惯、优美简捷的动机。

switch 是多分支语句，它是 if 语句的一个补充，但并不是必需的，当用它编制程序会带来可读性良好的效果时，就采用它。

用这些语句编制的程序，其结构性比较好，所以可读性也比较好。现代程序设计反对用goto编程，因为它破坏程序过程中的结构，使之不可读，难维护。

求 π 是常规的级数求和问题，任何级数的求和求积，都可进行类似的简单分析和实现。

判明素数以及求一定数域的素数分析，能够帮助读者认识到，算法能够相当程度地影响程序运行的效率。

求积分问题是要说明解决问题的算法很重要，而这种算法的分析设计并非易事，故并不是程序设计中学习的内容。

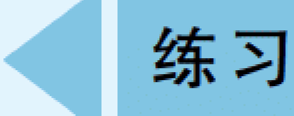
程序设计有3种方法。过程化方法最直接和简明，但它只能解决一些小问题。

对一个具体算法问题，编制程序相对较难，是因为涉及计算方法问题。利用现有的数学方法和结论，有助于问题的解决。

学习程序设计的主要任务是学习如何组织程序，表达实际问题的已有解决方法，而不是去寻找实际问题的解决方法。

寻找实际问题的解决方法属于系统分析与设计的范畴。本书介绍的只是常规和简单的问题求解方法和思路，使之通过程序设计分析与实现的经验积累，产生学习和掌握程序设计(面向对象程序设计)方法的向往。

程序设计更多的是体现其艺术性，可读性是我们追求的重要目标。



4.1 计算级数

$$1+x-\frac{x^2}{2!}+\frac{x^3}{3!}-\cdots+(-1)^{n+1}\frac{x^n}{n!}$$

要求精度为 10^{-8}。并分别用do…while、while和for语句编写程序。

4.2 编程求 $1!+2!+3!+4!+\cdots+12!$

4.3 编程求“水仙花数”。所谓“水仙花数”，是指一个三位数，其各位数字立方和等于该数本身。例如，153是水仙花数，因为 $153=1^3+5^3+3^3$。

4.4 编程求1000之内的所有“完数”。所谓“完数”，是指一个数恰好等于它的因子之和。例如，6是完数，因为 $6=1+2+3$。

4.5 一球从100m高度落下，每次落地后反跳回原高度的一半，再落下。编程求它在第10次落地时，共经过多少米？第10次反弹多高？

4.6 猴子吃桃问题。猴子第一天摘下若干桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想再吃时，见只剩一个桃子了。编程求第一天共摘下多少桃子。

4.7 用迭代法编程求 $x=\sqrt{a}$。

求平方根的迭代公式为：

$$x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$$

要求前后两次求出的x的差的绝对值小于 10^{-7}。



4.8　用循环语句编程打印如下图案：

（1）

```
                  #
                 # # #
                # # # # #
               # # # # # # #
              # # # # # # # # #
             # # # # # # # # # # #
            # # # # # # # # # # # # #
           # # # # # # # # # # # # # # #
          # # # # # # # # # # # # # # # # #
         # # # # # # # # # # # # # # # # # # #
```

（2）

```
# # # # # # # # # # # # # # # # #
 # # # # # # # # # # # # # # # #
  # # # # # # # # # # # # # # #
   # # # # # # # # # # # # # #
    # # # # # # # # # # # # #
     # # # # # # # # # # # #
      # # # # # # # # # # #
       # # # # # # # # # #
```

4.9　编程打印乘法九九表：

（1）

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

(2)

*	1	2	3	4	5	6	7	8	9
1	1								
2	2	4							
3	3	6	9						
4	4	8	12	16					
5	5	10	15	20	25				
6	6	12	18	24	30	36			
7	7	14	21	28	35	42	49		
8	8	16	24	32	40	48	56	64	
9	9	18	27	36	45	54	63	72	81

(3)

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2		4	6	8	10	12	14	16	18
3			9	12	15	18	21	24	27
4				16	20	24	28	32	36
5					25	30	35	40	45
6						36	42	48	54
7							49	56	63
8								64	72
9									81

4.10 编程求解问题。若一头小母牛，从出生起第四个年头开始每年生一头母牛，按此规律，第 n 年时有多少头母牛？

第5章 函数

要编好程序，就要会合理地划分程序中的各个程序块，C++称之为函数。函数有各种表现形态，但都离不开函数调用的实质。所以要用好函数，必须先把握函数调用机制。学习本章后，要求领会函数调用的内部实现机制，区分函数声明与定义，掌握全局变量、静态局部变量和局部变量之间的区别，理解并运用递归、内联、重载和默认参数的函数。

5.1 函数概述

程序通常是非常复杂而冗长的。实际编程中，有些程序需要几万甚至几百万行的代码。在编写一个很长的程序时，可以采用一种好的策略，就是把这个大的程序分割成一些相对独立而且便于管理和阅读的小块程序。这样，无论对程序员还是其他阅读者都很方便。

把相关的语句组织在一起，并给它们注明相应的名称，利用这种方法把程序分块，这种形式的组合就称为函数。函数通常也称为例程或过程。

函数的使用是通过函数调用实现的。函数调用指定了被调用函数的名字和调用函数所需的信息（参数），这和请一个上门服务的修理工形式类似。主人（相当于调用函数）要求修理工人（相当于被调用函数）按照要求（函数参数）完成某个任务，并在完成这项工作后由主人验收（函数返回）。如果不符合要求，则修理工人就面临拿不到工钱的局面。

程序员编写完成指定任务的函数是用户定义的函数，标准库函数是 C++提供的可以在任何程序中使用的公共函数。程序总是从 main()函数开始启动。

可以通过结合已有函数的方法建立新的函数。由多个小函数建立大函数，这能使程序易写、易读和易调试。

图 5-1 反映了 main()函数用层次式管理方式与被调用函数的关系。一个函数可以被函数调用也可以调用函数。

C++不允许函数定义嵌套，即在函数定义中再定义一个函数是非法的。

例如，下面的代码在主函数中非法嵌套了一个 func()函数定义：

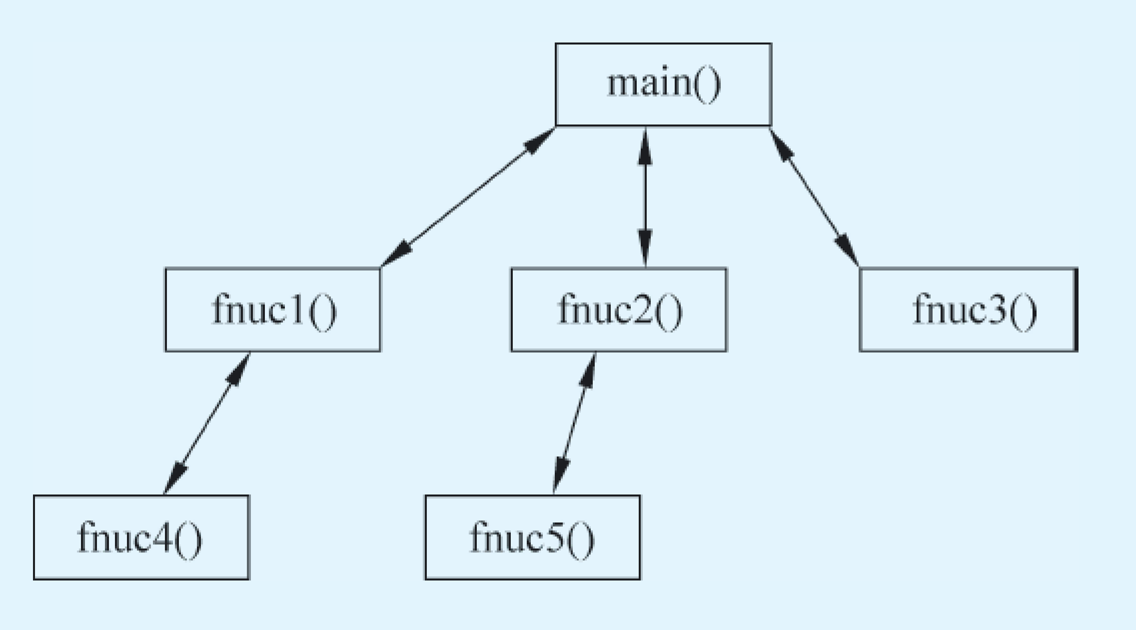


图 5-1　调用与被调用函数的层次关系

```
int main()
{
   void func()
   {
      //...
   }
}
```

C++函数是一个独立完成某个功能的语句块，函数与函数之间通过输入参数和返回值（输出）来联系。可以把函数看作是一个“黑盒(black box)”，除了输入、输出，其他什么都看不见。

例如我们只需了解怎样连接电源和天线及按钮操作，就能看到电视节目（输入与输出），而电视机内部的电子元件如何工作无须我们操心，这样的电子元件就称为“黑盒”。

函数的类型：

（1）获取参数并返回值，例如：

```
int bigger(int a, int b)
{
   return (a > b) ? a : b;
}
```

（2）获取参数但不返回值，例如：

```
void delay(long a)
{
   for(int i = 1; i <= a; i ++ ); //延迟一个小的时间片
}
```

（3）没有获取参数但返回值，例如：

```
int geti() //从键盘上获取一个整型数
{
   int x;
   cout <<"please input a integer:\n";
   cin >> x;
   return x;
}
```

（4）没有获取参数也不返回值，例如：

```
void message()      //在屏幕上显示一条消息
{
```

```
    cout <<"This is a message.\n"
}
```

5.2 函数原型

标准库函数的函数原型都在头文件中提供,程序可以用#include 指令包含这些原型文件。对于用户自定义函数,程序员必须在源代码中说明函数原型。

函数原型是一条程序语句,即它必须以分号结束。它由函数返回类型、函数名和参数表构成,形式为:

```
返回类型 function(参数表);
```

参数表包含所有参数的数据类型,参数之间用逗号分开。**在 C++ 中,函数声明就是函数原型**。

函数原型和函数定义在返回类型、函数名和参数表上必须完全一致。如果它们不一致,就会发生编译错误。

函数原型不必包含参数的名字,而只要包含参数的类型。下面的函数原型声明是合法的。

```
int Area(int,int);
```

等价于:

```
int Area(int length,int width);
```

对于标准库函数(简称库函数)来说,编译器从来不把其实际代码看成是程序的组成部分。编译器能够确认是否正确地调用库函数,这是必要的。在头文件中内含的函数声明都是函数原型。

如果函数原型不正确,编译器会及时报告错误。

例如,对于程序 ch1_3.cpp,主函数中的函数调用写成:

```
c = max(a,b,56);     //error:extra parameter in function call
```

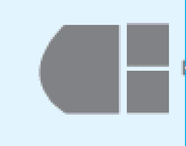
则编译器将会报告一个“函数调用中遇到过多的函数参数”的错误。

又如,下面的代码中,函数声明与函数定义的函数原型不一致:

```
void funcA(int,float);

int main()
{
    int a;
    float b;
    funcA(a,b);
}

void funcA(int,int)
{
    //...
}
```



该代码能够正确通过编译，因为函数声明的原型与函数调用相吻合。但在连接时，发现没有与函数声明相一致的函数定义，结果产生“不能确定的外部函数”的连接错误。

函数返回在声明时约定数据类型。例如：

```
int max(int a, int b)
{
   if(a > b)
     return a;
   else
     return b;
}
```

例中，函数返回的变量 a 或 b 是 int 型的。

如果返回的是其他基本数据类型，则在返回时，先作隐含的类型转换，然后再返回。例如，下面的代码中，主函数中的变量 a 被初始化为 3：

```
int f()
{
   return 3.5;
}

int main()
{
   int a = f();
}
```

因为函数 f()定义的返回类型是 int，所以 return 语句的值 3.5 被转换成 int 型数 3 之后，返回给主函数，赋给了变量 a。

如果函数返回的是不相容的数据类型(比如，后面介绍的类对象)，则函数将在编译时给出一个“不能将类转换成 int”的错误。

函数的返回值也称函数值。返回的不是函数本身，而是一个值。

return 语句后面的括号是任选的，例如，“return (3);”等价于“return 3;”。

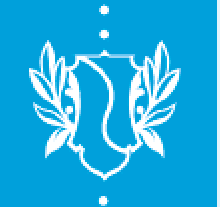
return 语句可用于改变执行顺序。例如：

```
int min(int a, int b)
{
   if(a < b)
     return a;
   else
     return b;
}
```

在这里，return 语句还起到了改变计算顺序的作用。因为 return 是返回语句，它将退出函数体，所以该语句之后的语句不会被执行了。

无返回的函数也可以使用 return，但不能返回值。例如：

```
void message(int a)
{
   if(a > 10)
```




```
        return;
    //...
}
```

在这里,return 语句起了一个改变语句顺序的作用。

在有返回类型的函数中,return 后面所跟的表达式并不直接替换调用函数,而是先经过求值计算,必要的时候进行类型转换,然后将其存放到内存的某个区域。该区域视编译器的不同而不同,也视返回类型的不同而不同。例如,BCB 将一个返回整型的数存放在栈区(见 5.4 节)的位置。在存放到一个专用的区域后,再将其变量的地址传给调用函数,以使调用函数把它作为函数返回值。

编译器遇到一个函数调用时,需要判断该函数调用是否正确,该机制即函数原型。

5.3 全局变量与局部变量

1. 程序的内存区域

并不是所有的变量时时刻刻都是可知的。一些变量在整个程序中都是可见的,它们称为全局变量。一些变量只能在一个函数中可知,称为局部变量。要了解变量的这些属性,应先弄清程序在内存中的分布区域,见图 5-2。

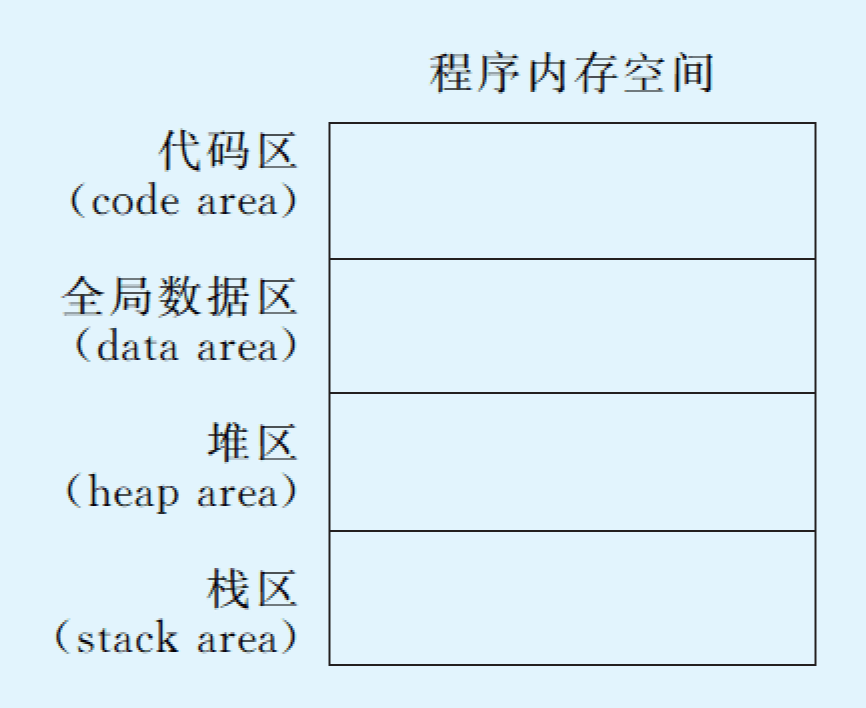


图 5-2 程序在内存中的区域

一个程序将操作系统分配给其运行的内存块分为 4 个区域:

(1) 代码区,存放程序的代码,即程序中的各个函数代码块。

(2) 全局数据区,存放程序的全局数据和静态数据。

(3) 堆区,存放程序的动态数据。

(4) 栈区,存放程序的局部数据,即各个函数中的数据。

2. 全局变量

在函数外边访问的变量被认为是全局变量,并在程序的每个函数中是可见的。全局变量存放在内存的全局数据区。全局变量由编译器建立,并且初始化为 0,在定义全局变量时,进行专门初始化的除外。

例如，下面的代码定义并使用了全局变量 n：

```
int n = 5;     //全局变量

int main()
{
   int m = n;
   //...
}
void func()
{
   int s;
   n = s;
   //...
}
```

n 在任何函数的外部定义。n 被初始化为 5，如果 n 不在定义时初始化，则 C++将其初始化为 0。main()函数使用变量 n，func()函数修改变量 n。两个函数都访问了同一个内存区域。这样定义的全局变量 n 在所有函数中都可见。如果一个函数修改了 n，则所有其他的函数都会看到修改后的变量。

全局变量在主函数 main()运行之前就开始存在了。所以主函数中可以访问 n 变量。全局变量通常在程序顶部定义。全局变量一旦定义后就在程序的任何地方可知。可以在程序中间的任何地方定义全局变量，但要在任何函数之外。**全局变量定义之前的所有函数定义，不会知道该变量**。例如：

```
int main()
{
   int m = n;          //error: n 无定义
   //...
}

int n;                 //全局变量

void func()
{
   int s;
   n = s;
   //...
}
```

该代码中的全局变量 n 不能被主函数 main()访问。编译该代码，将会引起 main()中的 m 初始化语句报告一个“n 无定义”的错误。

3. 局部变量

在函数内部定义的变量仅在该函数内是可见的。另外，局部变量的类型修饰是 auto，表示该变量在栈中分配空间，但习惯上都省略 auto。例如：

```
int main()
{
   int n;          //等价于 auto int n;
   //...
```




```
}

void func()
{
   int n;
   //...
}
```

代码中两个函数都包含一个变量定义语句。在函数内定义的变量局部于该函数。main()函数中有一个变量 n,func()函数中也有一个变量 n,但它们是两个不同位置的变量。

一个函数可以为局部变量定义任何名字,而不用担心其他函数使用过同样的名字。这个特点和局部变量的存在性使 C++适合于由多个程序员共同参与的编程项目。项目管理员为程序员指定编写函数的任务,并为程序提供参数和期望的返回值。然后,程序员着手编写函数,而不用了解程序的其他部分和项目中其他程序员所使用的变量名。

函数中的局部变量存放在栈区。在函数开始运行时,局部变量在栈区被分配空间;函数退出时,局部变量随之消失。

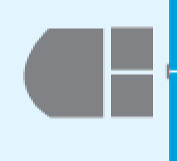
局部变量没有默认初始化。如果局部变量不被显式初始化,那么,其内容是不可预料的。例如:

```
//----------------------
//     ch5_1.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int func1();
int func2();
//----------------------
int main(){
  func1();
  cout << func2()<< endl;
}//---------------------
int func1(){
  int n = 12345;
  return n;
}//---------------------
int func2(){
  int m;
  return m;        //警告:m 可能没赋值就使用了
}//---------------------
```

运行结果为:

```
12345
```

主函数 main()先后调用了函数 func1()和 func2(),它们都是无参并返回整数的函数。在 func1()中,定义了局部变量 n,并给其初始化为 12345。在 func2()中,定义了局部变量 m,没有初始化。可是在将该变量值返回后,在主函数中输出该值,却发现为 12345,恰好就是 func1()函数中初始化的值。这说明 func2()中没有显式初始化的局部变量 m,C++也未给其默认初始化,其值保留为原内存位置的值。那么,原内存位置为什么恰巧是存放值 12345 的位置呢? 请见 5.4 节“函数调用机制”。



5.4 函数调用机制

栈是一种数据结构，它的工作原理就像在子弹匣中压子弹一样，最先压入的子弹要等到最后才飞射出去，而最后压入的子弹则首先飞射出去。

C++的函数调用过程需要调用初始化和善后处理的环节。函数调用的整个过程就是栈空间操作的过程。函数调用时，C++首先：

(1) 建立被调函数的栈空间。

(2) 保护调用函数的运行状态和返回地址。

(3) 传递参数。

(4) 将控制转交给被调函数。

例如，下面的代码在主函数中调用一个函数，该函数又调用了另一个函数，它得到图 5-3 中所示的内存布局：

```
void funcA(int,int);
void funcB(int);

int main()
{
    int a = 6,b = 12;
    funcA(a,b);
}

void funcA(int aa,int bb)
{
    int n = 5;
    //...
    funcB(n);
}

void funcB(int s)
{
    int x;
    //...
  }
```

该函数运行的栈区分布图是示意性的。**函数的栈操作原理是一致的，但具体的实现因编译器不同而不同**。介绍函数栈的具体操作过程的目的是要让读者对函数运行的机制有一个感性的认识。

在图 5-3 中，主函数 main()的返回地址是在操作系统的内存驻留地中。其参数是操作系统传递过来的，在 8.9 节中将对其进行介绍。主函数定义了两个局部变量 a 和 b，a 和 b 的次序并不一定如图 5.3 所示，这些实现的细节都是与 C++标准无关的。我们观察栈内存时，可以忽略返回地址和调用函数运行状态。

当主函数调用 funcA()时，funcA()着手保护调用函数的地址等数据，分配两个形参的空间，将主函数的实参传递过来。所以 funcA()中的形参 aa 和 bb 分别为 6 和 12。

funcA()的栈区和 main()的栈区是互相独立的。在 funcA()中不能访问 main()中的局部变量 a 和 b。通过参数传递，使得 funcA()中的形参赋有 main()函数中的实参值。funcA()可

以修改其变量 aa 和 bb,但始终不会影响 main()中的 a 和 b。这便是 C++函数的参数传值特性。

	栈区
	……
x	
s	5
funcB()	返回地址
	调用函数运行状态
n	5
bb	12
aa	6
funcA()	返回地址
	调用函数运行状态
b	12
a	6
main()	参数
	返回地址
	操作系统运行状态

图 5-3　函数调用机制中的栈结构

在函数 funcA()中定义了一个局部变量 n,并以该变量作实参来调用函数 funcB()。同样,funcB()建立了它自己的栈空间,保护 funcA()的返回地址,获取参数值,随后运行它自己的函数体语句。

栈是有限的资源,每次嵌套调用一个函数,剩余的栈空间会逐渐减少。可以看出,如果一层层地调用下去,或者过多地定义局部变量,特别是数组(将在第 7 章介绍),最后可能导致栈空间枯竭而引起程序运行出错。如果程序确实要占用相当大的栈空间,可以在连接前通过设置栈空间大小来改善。

函数在返回时,将把返回值保存在临时变量空间中(如果有返回值的话)。然后恢复调用函数的运行状态,释放栈空间,使其属于调用函数栈空间的一部分,最后根据返回地址,回到调用函数代码执行处。

图 5-4 是 funcB()返回到 funcA(),funcA()又回到 main()时的栈内容。可以看出,返回到主函数后,funcA()和 funcB()的局部变量和形参都消失,但对应内存中的内容还是存在着,这就是 5.3 节中程序 ch5_1.cpp 得到该运行结果的原因。

	栈区
	……
	5
	返回地址
	调用函数运行状态
	5
	12
	6
	返回地址
	调用函数运行状态
b	12
a	6
main()	参数
	返回地址
	操作系统运行状态

图 5-4　函数返回到 main()时的内存情况

5.5 静态局部变量

在局部变量前加上“static”关键字，就成了静态局部变量。静态局部变量存放在内存的全局数据区。函数结束时，静态局部变量不会消失，每次调用该函数时，也不会为其重新分配空间。它始终驻留在全局数据区，直到程序运行结束。静态局部变量的初始化与全局变量类似，如果不为其显式初始化，则C++自动为其初始化为0。

静态局部变量与全局变量共享全局数据区，但静态局部变量只在定义它的函数中可见。静态局部变量与局部变量在存储位置上不同，使得其存在的时限也不同，导致对二者操作的运行结果也不同。

例如，下面的程序定义了全局变量、静态局部变量和局部变量：

```
//----------------------
//    ch5_2.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void func();
void print(int a, int b, int n);
int n = 1;            //全局变量
//----------------------
int main(){
  static int a;       //静态局部变量
  int b = -10;        //局部变量
  print(a,b,n);
  b += 4;
  func();
  print(a,b,n);
  n += 10;
  func();
}//--------------------
void func(){
  static int a = 2;   //静态局部变量
  int b = 5;          //局部变量
  a += 2;
  n += 12;
  b += 5;
  print(a,b,n);
}//--------------------
void print(int a, int b, int n){
  cout <<"a:"<< a
      <<" b:"<< b
      <<" n:"<< n << endl;
}//--------------------
```

运行结果为：

```
a:0 b:-10 n:1
a:4 b:10 n:13
```




```
a:0 b:-6 n:13
a:6 b:10 n:35
```

程序中主函数 main()两次调用了 func()函数,从运行结果可以看出,程序控制每次进入 func()函数时,局部变量 b 都被初始化。而静态局部变量 a 仅在第一次调用时被初始化,第二次进入该函数时,不再进行初始化,这时它的值是第一次调用后的结果值 4。main()函数中的变量 a 和 b 与 func()函数中的变量 a 和 b 空间位置是不一样的,所以相应的值也不一样。关于变量作用域和可见性的进一步讨论见第 6 章。

静态局部变量的用途有很多:可以使用它确定某函数是否被调用过;使用它保留多次调用的值。

5.6 递归函数

1. 什么是递归函数

递归函数(recursive function)即自调用函数,在函数体内部直接或间接地自己调用自己,即函数的嵌套调用是函数本身。

例如,下面的程序为求 n!:

```
long fact(int n)
{
   if(n == 1)
    return 1;

   return fact(n-1) * n;     //出现函数自调用
}
```

2. 函数调用机制的说明

任何函数之间不能嵌套定义,调用函数与被调用函数之间相互独立,但彼此可以调用。

发生函数调用时,被调函数中保护了调用函数的运行环境和返回地址,使得调用函数的状态可以在被调函数运行返回后完全恢复,而且该状态与被调函数无关。

被调函数运行的代码虽是同一个函数的代码体,但由于调用点、调用时状态、返回点的不同,可以看作是函数的一个副本,与调用函数的代码无关,所以函数的代码是独立的。

被调函数运行的栈空间独立于调用函数的栈空间,所以与调用函数之间的数据也是无关的。函数之间靠参数传递和返回值来联系,函数看作为黑盒。

这种机制决定了 C/C++允许函数递归调用。

3. 递归调用的形式

递归调用有直接递归调用和间接递归调用两种形式。

直接递归即在函数中出现调用函数本身。

例如,下面的代码求斐波那契数列第 n 项。斐波那契数列的第一项和第二项是 1,后面每一项是前两项之和,即 1,1,2,3,5,8,13,…。代码中采用直接递归调用:

```
long fib(int x)
{
   if(x>2)
     return (fib(x-1)+fib(x-2));   //直接递归
   else
     return 1;
}
```

间接递归调用是指函数中调用了其他函数，而该函数却又调用了本函数。

例如，下面的代码定义两个函数，它们构成了间接递归调用：

```
int fn1(int a)
{
   int b;
   b=fn2(a+1);     //间接递归
       //...
}

int fn2(int s)
{
   int c;
   c=fn1(s-1);     //间接递归
       //...
}
```

本例中，fn1()函数调用了 fn2()函数，而 fn2()函数又调用了 fn1()函数。需要注意的是，fn2 函数必须在 fn1 函数之前预先声明。

4. 递归的条件

(1) 须有完成函数任务的语句。

例如，下面的代码定义了一个递归函数：

```
#include <iostream>
using namespace std;
void count(int val)              //递归函数可以没有返回值
{
   if(val>1)
       count(val - 1);

   cout <<"ok:" << val << endl;   //此语句完成函数任务
}
```

该函数的任务是在输出设备上显示“ok:整数值”。

(2) 一个确定是否能避免递归调用的测试。

例如，上例的代码中，语句“if(val>1)”便是一个测试，如果不满足条件，就不进行递归调用。

(3) 一个递归调用语句。

该递归调用语句的参数应该逐渐逼近不满足条件，以致最后断绝递归。

例如，上面的代码中，“count(val − 1);”是一个递归调用，参数值在渐渐变小，这种发展趋势能使测试“if(val>1)”最终不满足。

(4) 先测试,后递归调用。

在递归函数定义中,必须先测试,后递归调用。也就是说,递归调用是有条件的,满足了条件后,才可以递归。

例如,下面的代码无条件调用函数自己,造成无限制递归,终将使栈空间溢出:

```
#include <iostream>
using namespace std;
void count(int val)
{
   count(val - 1);      //无限制递归
   if(val > 1)          //该语句无法到达
      cout << "ok:" << val << endl;
}
```

5. 消去递归

理论上已经证明,递归函数都能用非递归函数来代替。存在非递归化的规范方法,此处不予展开。例如,下面的代码求两个整数 a、b 的最大公约数,用递归和非递归函数分别定义之:

```
long gcd1(int a, int b)    //递归版
{
   if(a % b == 0)
      return b;
   return gcd1(b, a % b);
}

long gcd2(int a, int b)    //非递归版
{
   int temp;
   while(b!= 0)
   {
      temp = a % b;
      a = b;
      b = temp;
   }
   return a;
}
```

思考:将求 n! 的递归函数非递归化。

6. 递归的评价

递归的目的是简化程序设计,使程序易读。

但递归增加了系统开销。时间上,执行调用与返回的额外工作要占用 CPU 时间。空间上,随着每递归一次,栈内存就多占用一截。

相应的非递归函数虽然效率高,但却比较难编程,而且相对来说可读性差。

现代程序设计的目标主要是可读性好。随着计算机硬件性能的不断提高,程序在更多的场合强调可读性,它似乎在鼓励递归设计。但是,递归函数如果很缓慢地逼近到递归结束条件,会使性能大大下降,所以实际上只有很少一部分类似于最大公约数计算的递归设计才被接受。

5.7 内联函数

1. 内联函数的需要性

内联函数也称内嵌函数,它主要是解决程序的运行效率。

函数调用需要建立栈内存环境,进行参数传递,并产生程序执行转移,这些工作都需要一些时间开销。

有些函数使用频率高,但代码却很短。

例如,下面的代码中,频繁地调用一个小函数:

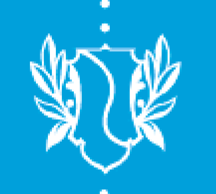
```
//----------------------
//    ch5_3.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int isnumber(char);           //函数声明
//----------------------
int main(){
  char c;
  while((c=cin.get())!='\n')
    if( isnumber(c) )         //调用一个小函数
      cout <<"you entered a digit\n";
    else
      cout <<"you entered a non_digit\n";
}//---------------------
int isnumber(char ch){        //函数定义
  return (ch>='0' && ch<='9')? 1:0;
}//---------------------
```

程序中不断到设备中读取数据,频繁调用 isnumber()函数。为了提高效率,可将程序改为:

```
#include <iostream>
using namespace std;
int main()
{
   char c;
   while((c=cin.getc())!='\n')
   {
     if((ch>='0' && ch<='9')? 1:0 )      //修改处:直接计算表达式
       cout <<"you entered a digit\n";
     else
           cout <<"you entered a non-digit\n";
   }
}
```

该程序在 if 语句中用表达式替换了函数调用。在程序运行上提高了一些执行效率,因为免去了大量的函数调用开销。

由于 isnumber()比相应的表达式可读,所以若程序中多处出现 isnumber()的替换,就



会降低程序的可读性。我们既要用函数调用来体现其结构化和可读性,又要使效率尽可能地高。

2. 解决办法

将 isnumber()函数声明为 inline,即在函数声明和定义中:

```
inline int isnumber(char);

inline int isnumber(char c)
{
   return (ch>='0'&& ch<='9')?1:0;
}
```

编译器看到 inline 后,为该函数创建一段代码,以便在后面每次碰到该函数的调用都用相应的一段代码来替换。内联函数可以在一开始仅声明一次。例如下面的代码表达了一个内联函数:

```
#include<iostream>
using namespace std;
inline int isnumber(char);     //inline 函数声明

int main()
{
   char c;
   while((c=cin.getc())!='\n')
   {
      if( isnumber(c) )          //调用一个小函数
         cout<<"you entered a digit\n";
      else
         cout<<"you entered a non-digit\n";
   }
}

int isnumber(char ch)           //此处无 inline,视为 inline
{
   return (ch>='0'&& ch<='9')? 1:0;
}
```

3. 先声明后调用

内联函数必须在被调用之前声明或定义。因为内联函数的代码必须在被替换之前已经生成被替换的代码,因此,下面的代码不会像预计的那样被编译:

```
#include<iostream>
using namespace std;
int isnumber(char);        //此处无 inline

int main()
{
   char c;
   while((c=cin.getc())!='\n')
   {
      if( isnumber(c) )    //调用一个小函数
```

```
            cout <<"you entered a digit\n";
        else
            cout <<"you entered a non-digit\n";
    }
}

inline int isnumber(char ch)     //此处为 inline
{
    return (ch>='0'&& ch<='9')? 1:0;
}
```

编译程序不认为那是内联函数，对待该函数如普通函数那样，产生该函数的调用代码，并进行连接。

4. 内联函数的函数体限制

内联函数中不能含有复杂的结构控制语句，如 switch 和 while。如果内联函数有这些语句，则编译将该函数视同普通函数那样产生函数调用代码。

另外，递归函数（自己调用自己的函数）是不能被用来做内联函数的。

内联函数只适合于只有 1～5 行的小函数。对一个含有许多语句的大函数，函数调用和返回的开销占比相对来说微不足道，所以也没有必要用内联函数实现。

→ 内联函数与宏定义

在 C 中，常用预处理语句＃define 来代替一个函数定义。例如：

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

该语句使得程序中每个出现 MAX(a,b)函数调用的地方都被宏定义中后面的表达式((a)>(b)？(a)：(b))所替换。

宏定义语句的书写格式有过分的讲究，MAX 与括号之间不能有空格，所有的参数都要放在括号里。尽管如此，它还是有麻烦：

```
int a=1,b=0;
MAX(a++,b);            //a 被增值 2 次
MAX(a++,b+10);         //a 被增值 1 次
MAX(a,"Hello");        //错误地比较 int 和字符串，没有参数类型检查
```

MAX()函数的求值会由于两个参数值的大小不同而产生不同的副作用。

MAX(a++,b)的值为 2，同时 a 的值为 3；

MAX(a++,b+10)的值为 10，同时 a 的值为 2。

如果是普通函数，则 MAX(a,"Hello")会受到函数调用的检查，但此处不会因为两个参数类型不同而被编译拒之门外。幸运的是，通过一个内联函数可以得到所有宏的替换效能和所有可预见的状态以及常规函数的类型检查：

```
inline int MAX(int a, int b)
{
    return a>b?a:b;
}
```



5.8 重载函数

1. 重载的必要性

在 C 中，每个函数必须有其唯一的名字，但有时，这会令人生厌。

例如，求一个数的绝对值，由于命名唯一，所以对于不同的类型需要不同名字的函数：

```
int abs(int);
long labs(long);
double fabs(double);
```

这几个函数所做的事情是一样的，都是求绝对值。因此，使用 3 个不同的函数名，看上去很笨拙，若给以同样的名字就会方便得多。对于在不同类型上作不同运算而又用同样的名字的情况，则称之为重载。这种技术在 C++ 中早已用于基本数据类型运算，如加法只有一个操作符名字＋，但它可以用来加整数值、浮点值和指针值。

例如，上述 3 个函数的声明可以改为：

```
int abs(int);
long abs(long);
double abs(double);
```

C++用一种函数命名技术可以准确判断出应该使用哪个 abs()函数。例如：

```
abs(-10);              //调用 int abs(int);
abs(-1000000);         //调用 long abs(long);
abs(-12.23);           //调用 double abs(double);
```

2. 匹配重载函数的顺序

在调用一个重载函数 f()时，编译器必须搞清函数名 f 究竟是指哪个函数。这是靠将实参类型和所有被调用的 f()函数的形参类型一一比较来判定的。按下述 3 个步骤的先后顺序找到并调用那个函数：

(1) 寻找一个严格的匹配，如果找到了，就用那个函数。

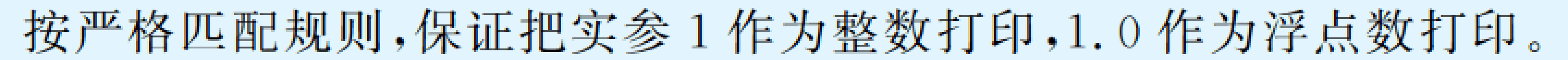
(2) 通过内部转换寻求一个匹配，只要找到了，就用那个函数。

(3) 通过用户定义的转换寻求一个匹配，若能查出有唯一的一组转换，就用那个函数(见 18.5 节中的后增量函数 type operator ++(type&,int))。

例如，重载函数 print()的匹配：

```
void print(double);
void print(int);

void func()
{
   print(1);          //匹配 void print(int);
   print(1.0);        //匹配 void print(double);
   print('a');        //匹配 void print(int);
   print(3.1415f);    //匹配 void print(double);
}
```



按严格匹配规则，保证把实参 1 作为整数打印，1.0 作为浮点数打印。

对于 int 形参，0、char 和 short int 都是严格匹配。

对于 double 形参，float 也是严格匹配。

C++允许 int 到 long，int 到 double 的转换。当实参是整数，而重载函数一为 long 型参数，一为 double 型参数时，应该给以一个显式转换。

例如，对于重载函数 print()声明，其下面的函数调用将引起错误：

```
void print(long);
void print(double);

void func(int a)
{
   print(a);        //error: 因为有二义性
}
```

该代码在 BC 中会导致二义性编译错误(Ambiguity between "print(long)" and "print(double)")，应该指明是“print(long(a));”还是“print(double(a));”，以分辨这种含混的调用。

3. 使用说明

(1) C++的函数如果在返回类型、参数类型、参数个数、参数顺序上有所不同，则认为是不同的。但重载函数如果仅仅是返回类型不同，则是不够的。

例如，下面的声明是错误的：

```
void func(int);
int func(int);
```

编译器无法区分函数调用“func(3)”是指上述哪一个重载函数。因此**重载函数至少在参数个数、参数类型或参数顺序上有所不同**。

(2) typedef 定义的类型只能使之相同于一个已存在的类型，而不能建立新的类型，所以不能用 typedef 定义的类型名来区分重载函数声明中的参数。

例如，下面的代码实际上是同一个函数：

```
typedef INT int;

void func(int x) { //… }
void func(INT x) { //… }     //error: 函数重复定义
```

编译器不能区分这两个函数的差别，INT 只不过是 int 的另一种称呼而已。

(3) 让重载执行不同的功能，是不好的编程风格。同名函数应该具有相同的功能。如果定义一个 abs()函数而返回的却是一个数的平方根，则该程序的可读性受到破坏。

→ 关于重载函数的内部实现

C++用名字粉碎(name mangling)的方法来改变函数名，以区分参数不同的同名函数。名字粉碎是十分简单的过程，一系列代码被附加到函数名上以指示参数类型以及它们出现的次序。例如，用 v、c、i、f、l、d、r 分别表示 void、char、int、float、long、double、long double，则重载函数：

```
int f(char a);
int f(char a, int b, double c);
```

在内部分别被表示为 f_c 和 f_cid,而函数:

```
int f_cid();
```

则在内部表示为 f_cid_v。它与上面的重载函数相区别。

名字粉碎是看不到的,它通常放在目标文件中,遇到连接错误时,有时可能会看到粉碎了的名字。

5.9 默认参数的函数

1. 默认参数的目的

C++可以给函数定义默认参数值。通常,调用函数时,要为函数的每个参数给定对应的实参。例如:

```
void delay(int loops);        //函数声明

void delay(int loops)         //函数定义
{
   if(loops == 0)
         return;
   for(int i = 0; i < loops; i ++ );
}
```

无论何时调用 delay()函数,都必须给 loops 传一个值以确定时间。但有时需要用相同的实参反复调用 delay()函数。C++可以给参数定义默认值。如果将 delay()函数中的 loops 定义成默认值 1000,只需简单地把函数声明改为:

```
void delay(int loops = 1000);
```

这样,无论何时调用 delay()函数,都不用给 loops 赋值,程序会自动将它当作值 1000 进行处理。例如,调用:

```
delay(2500);       //loops 设置为 2500
delay();           //ok: loops 采用默认值 1000
```

调用中,若不给出参数,则按指定的默认值进行工作。

允许函数默认参数值是为了让编程简单,让编译器做更多的错误检查工作。

2. 默认参数的声明

默认参数在函数声明中提供,**当又有声明又有定义时,定义中不允许默认参数**。如果函数只有定义,则默认参数才可出现在函数定义中。例如:

```
void point(int = 3, int = 4);      //声明中给出默认值

void point(int x, int y)           //定义中不允许再给出默认值
```

```
    {
        cout << x << endl;
        cout << y << endl;
    }
```

3. 默认参数的顺序规定

如果一个函数中有多个默认参数，则形参分布中，默认参数应从右至左逐渐定义。当调用函数时，只能向左匹配参数。例如：

```
void func(int a = 1, int b, int c = 3, int d = 4);      //error

void func(int a, int b = 2, int c = 3, int d = 4);      //ok
```

对于第 2 个函数声明，其调用的方法规定为：

```
func(10,15,20,30);          //ok：调用时给出所有实参
func();                     //error：参数 a 没有默认值

func(12,12);                //ok：参数 c 和 d 默认

func(2,15,,20);             //error：只能从右到左顺序匹配默认
```

4. 默认参数与函数重载

默认参数可将一系列简单的重载函数合成为一个。例如，下面 3 个重载函数：

```
void point(int,int) {   //... }
void point(int a) { return point(a,4); }
void point() { return point(3,4); }
```

可以用下面的默认参数的函数来替代：

```
void point(int = 3,int = 4);
```

当调用“point();”时，即调用“point(3,4);”，它是第 3 个声明的重载函数。

当调用“point(6);”时，即调用“point(6,4);”，它是第 2 个声明的重载函数。

当调用“point(7,8);”时，即调用第 1 个声明的重载函数。

如果一组重载函数(可能带有默认参数)都允许相同实参个数的调用，将会引起调用的二义性。例如：

```
void func(int);             //重载函数之一
void func(int,int = 4);     //重载函数之二：带有默认参数
void func(int = 3,int = 4); //重载函数之三：带有默认参数

func(7);                    //error：到底调用 3 个重载函数中的哪个?
func(20,30)                 //error：到底调用后面 2 个重载函数中的哪个?
```

5. 默认值的限定

默认值可以是全局变量、全局常量，甚至是一个函数。例如：




```
int a = 1;
int fun(int);

int g(int x = fun(a));      //ok: 允许默认值为函数
```

默认值不可以是局部变量,因为默认参数的函数调用是在编译时确定的,而局部变量的位置与值在编译时均无法确定。例如:

```
void fun()
{
   int i;
   void g(int x = i);     //error:处理 g()函数声明时,i 不可见
}
```

小结

随着程序量和程序复杂度的不断增加,最好的办法是把程序分成更小、更容易管理的模块,这种模块就是函数。

函数名最好能反映出所要完成的任务。

函数可以把数据返回给调用者,若函数要返回一个值,必须在函数名前规定返回值的类型;若函数没有返回值,则类型为 void。

程序通过参数把信息传递给函数,若函数需要接受参数,就必须给参数指定名称及类型。

C++必须知道函数的返回类型以及接受的参数个数和类型,如果函数的定义出现在函数调用之后,就必须在程序的开始部分用函数原型进行说明。

局部变量是在函数内部定义的,只能被定义该变量的函数访问。全局变量是指其作用域贯穿程序始终的变量。定义全局变量要在程序开始时进行,并且放在所有函数的外面。静态局部变量是在函数内部定义,但生命期却随函数的第一次被调用而产生,随程序的结束而结束,静态局部变量只能在定义该变量的函数中可见。

函数调用机制是由栈操作的过程实现的。函数可以递归调用。函数定义不能放在任何函数定义的里面。

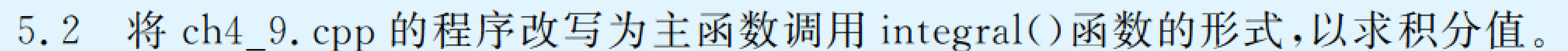
内联函数是为了提高编程效率而实现的,它克服了用#define 宏定义所带来的弊病。

函数重载允许用同一个函数名定义多个函数。连接程序会根据传递给函数的参数数目、类型和顺序调用相应的函数。函数重载使程序设计简单化,程序员只要记住一个函数名,就可以完成一系列相关的任务。

在函数定义中通过赋值运算,即可指定默认参数值。一旦程序在调用函数时默认了参数值,函数就使用默认参数值。不允许在参数中间使用默认值。指定默认参数值可以使函数的使用更为简单,同时也增强了函数的可重用性。

练习

5.1 将 ch4_8.cpp 的程序改写为主函数调用 isprime()函数的形式,以确定该数是否为素数。



5.2 将 ch4_9.cpp 的程序改写为主函数调用 integral()函数的形式，以求积分值。

5.3 将习题 4.9 打印乘法九九表改用函数调用的形式，适当取函数名，分别调用 3 种函数以输出不同格式。

5.4 分析下列程序，写出运行结果。

```
#include <iostream>
using namespace std;
//---------------------
void func();
int n = 1;
//---------------------
int main(){
  static int x = 5;
  int y = n;
  cout <<"Main -- x = "<< x
      <<", y = "<< y <<", n = "<< n << endl;
  func();
  cout <<"Main -- x = "<< x
      <<", y = "<< y <<", n = "<< n << endl;
  func();
}//---------------------
void func(){
  static int x = 4;
  int y = 10;
  x += 2;
  n += 10;
  y += n;
  cout <<"Func -- x = "<< x
      <<", y = "<< y <<", n = "<< n << endl;
}//---------------------
```

5.5 用非递归的函数调用形式求斐波那契数列第 n 项。

5.6 以下函数 poly 是用递归方法计算 x 的 n 阶勒让德多项式的值。已有调用语句“p(n,x);”，编写 poly 函数。递推公式如下：

```
polyn(x) = 1      当 n = 0 时;
polyn(x) = x      当 n = 1 时;
polyn(x) = ((2n - 1) * x * polyn-1(x) - (n - 1) * polyn-2(x))/n    当 n>1 时.
```

5.7 已知 $f(x)=\cos(x)-x$。x 的初始值为 3.14159/4，用牛顿法求解方程 $f(x)=0$ 的近似解，要求精确到 10^{-6}。f(x)的牛顿法为：

$$x_{n+1} = x_n - (\cos(x_n) - x_n)/(\sin(x_n) - 1)$$

5.8 编写程序，其中包含 3 个重载的 display()函数。第一个函数输出一个 double 值，前面用字符串“A double:”引导；第二个函数输出一个 int 值，前面用字符串“A int:”引导；第三个函数输出一个 char 字符，前面用字符串“A char:”引导。在主函数中，分别用 double、float、int、char 和 short 型变量去调用 display()函数，并对结果做简要说明。

5.9 将习题 4.10 用递归函数方法求解。

第6章　程序结构

要编好 C++程序，就必须要对 C++的程序结构有一个全面的了解。所有的 C++程序都是由一个或多个函数构成的。一个 C++程序可以由一个或多个包含若干函数定义的源文件组成。C++的编译器和连接器把构成一个程序的若干源文件有机地联络在一起，最终产生可执行程序。学习本章后，要求掌握外部存储类型和静态存储类型在多文件程序中的联络作用，理解作用域、可见性与生命期的概念，学会使用头文件，理解多文件结构，理解编译预处理的概念。

6.1　外部存储类型

一个程序在很小的规模下，可以用一个源文件来完整表达。本章之前示例的程序都是由单个文件构成的完整程序。一般具有应用价值的程序由多个源文件组成。根据 C++程序的定义，其中只有一个源文件具有主函数 main()，而其他的文件不能含有 main()，否则程序不知道该从何处开始执行了。

构成一个程序的多个源文件之间，通过声明数据或函数为外部的(extern)来进行沟通。例如，下面两个文件构成了一个程序，该程序由一个工程文件 ch6_1.prj 定义。各编译器对程序文件整合的工程管理和操作略有不同。工程文件和源文件中的内容分别为：

```
//*********************
// ch6_1.prj
// 作为 C++的 Console Application 工程，
// 不同的 C++软件有不同的设置方法，
// 但其所组合的 cpp 代码文件是统一的
//*********************
ch6_1.cpp
ch6_1_1.cpp
//*********************

//---------------------
//    ch6_1.cpp
//---------------------
```

```
#include <iostream>
using namespace std;
//-----------------------
void fn1();
void fn2();
int n;
//-----------------------
int main(){
  n = 3;
  fn1();                    //fn1()函数的定义在本文件中
  cout << n << endl;
}//---------------------
void fn1(){
  fn2();                    //fn2()函数的定义不在本文件中
}//---------------------

//-----------------------
//    ch6_1_1.cpp
//-----------------------
extern int n;               //n 由 ch6_1.cpp 定义
//-----------------------
void fn2(){                 //fn2()函数用于 ch6_1.cpp
  n = 8;                    //使用 n
}//---------------------
```

运行结果为：

```
c:\> ch6_1
8
```

文件 ch6_1.cpp 中含有主函数 main()，main()中调用了函数 fn1()，函数 fn1()调用了函数 fn2()，所以在 main()函数的前面应有函数 fn1()的声明，在函数 fn1()的定义之前应有函数 fn2()的声明。**所有函数声明一般都放在源文件的开始位置**。

文件 ch6_1.cpp 中定义了全局变量 n 以供 main()函数使用。

文件 ch6_1.cpp 中只含有函数 main()和 fn1()的定义，fn2()函数的定义在 ch6_1_1.cpp 中。

文件 ch6_1_1.cpp 中定义了函数 fn2()。函数 fn2()中要使用在 ch6_1.cpp 中定义的全局变量 n，为此在文件开头声明了带 extern 的 int n，它表示该变量 n 不在本文件中分配空间，而在程序的其他文件中分配空间(变量定义)。

默认的函数声明或定义总是 extern 的，所以文件 ch6_1.cpp 中，为了调用 fn1()和 fn2()，在文件一开始声明的函数原型等价于：

```
extern void fn1();
extern void fn2();
```

它们告诉连接程序，在所有组成该程序的文件(这里是 ch6_1.cpp 和 ch6_1_1.cpp)中搜索该函数的定义。其中，fn1()在本文件中定义，fn2()在 ch6_1_1.cpp 中定义。

假如一个程序由 10 个源文件构造而成，每个源文件都必须访问一个全局变量。在这种情形下，其中的 9 个文件必须把变量声明为 extern，另外一个则不能。虽然在包含 main()函数的源文件中分配变量是最合理的，但哪个文件真正分配该变量(全局变量定义)是无关紧要的。



带 extern 的变量说明是变量声明,不是变量定义。

如果共同的变量一次都没有定义,或者在各个文件中分别定义,造成定义多次,或者声明的类型不一致,都会造成直接或间接的错误。例如:

```
//file1.cpp

int a = 5;
int b = 6;
extern int c;

//file2.cpp

int a;                    //error: 多次定义
extern double b;          //error: 类型不一致
extern int c;             //error: 无定义
```

在上面的代码中,两个源文件都以常规方式定义变量 a,没有一个显式声明 extern,这时,其行为将依赖于编译器。VC 会通过编译,但在连接时,给出一个"变量 a 的定义已经在前一个文件中定义过"(int a already defined in file1. obj)的错误。但 BC 则将每个文件的变量定义都看作是全局静态变量(见 6.2 节)。所以,程序将会运行,但两个文件中的变量 a 是互不相干的。

两个文件中 b 的类型不一样,这时,VC 将在连接时报告一个"未定的外部名"(link error:unresolved external)错误,而 BC 却不能发现该错误而使程序错误地运行下去。

两个源文件又都声明变量 c 为 extern,这时,编译也不会发生问题,但连接时却会找不到该变量,产生一个连接错误,因为没有一个文件为该变量分配空间。

不论各编译器和连接器对外部存储类型的反应如何,只要理解了 extern 的作用,就可保证编程的正确性。

6.2 静态存储类型

1. 静态全局变量

在全局变量前加一个 static,使该变量只在这个源文件中可用,称之为全局静态变量。全局静态变量就是静态全局变量。

在前面几章,由于程序都是由一个源文件实现,所以全局变量与全局静态变量是没有区别的。但是在多文件组成的程序里,全局变量与全局静态变量是不同的。全局静态变量使得该变量成为由定义该变量的源文件所独享。

例如,两个源文件组成一个程序,其工程文件为 ch6_2. prj。其中一个源文件定义了"int n;",另一个源文件定义了"static int n;",则程序给它们分别分配了空间,两个值互不干扰:

```
//**********************
// ch6_2.prj
//**********************
ch6_2.cpp
ch6_2_1.cpp
//**********************
```

```
//----------------------
//    ch6_2.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
int n;
void fn();
//----------------------
int main(){
  n=20;
  cout<<n<<endl;
  fn();
}//---------------------

//----------------------
//    ch6_2_1.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
static int n;          //默认初始化为 0
//----------------------
void fn(){
  n++;
  cout<<n<<endl;
}//---------------------
```

运行结果为：

```
c:\>ch6_2
20
1
```

该程序分别编译后，连接为 ch6_2.exe。运行 ch6_2 便得到上述结果。

函数 fn()输出 1 而不是 21，表示两个变量互不干涉。

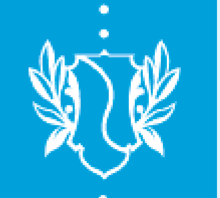
静态全局变量对组成该程序的其他源文件是无效的。

例如，下面的代码在 file1.cpp 中声明全局变量 n，在 file2.cpp 中定义全局静态变量 n：

```
//file1.cpp

#include<iostream>
using namespace std;
void fn();
extern int n;     //表示 n 仅是一个声明，将由别的文件定义
int main()
{
   n=20;
   cout<<n<<endl;
   fn();
}

//file2.cpp
```




```
#include <iostream>
using namespace std;
static int n;     //默认初始化为0

void fn()
{
   n++;
   cout << n << endl;
}
```

文件 file1.cpp 和 file2.cpp 分别都能通过编译,但连接时,file1.cpp 中的变量 n 找不到定义,产生连接错误。

如果把文件 file1.cpp 中的"extern int n;"去掉,那么编译会由于没有 n 的定义而出错。这说明 file2.cpp 中的静态变量 n 与 file1.cpp 中的变量 n 毫不相干。

使一个变量只在一个源文件中全局使用有时是必要的。第一,不必担心另外源文件使用它的名字,该名字在源文件中是唯一的;第二,源文件的全局变量不能被其他源文件所用,不能被其他源文件所修改,保证变量的值是可靠的。

2. 静态函数

函数的声明和定义默认情况下在整个程序中是外部(extern)的。有时候,你可能需要使某个函数只在一个源文件中有效,不能被其他源文件所用,这时在函数前面加上 static。

例如,下面的代码在 file2.cpp 中定义的函数不能被 file1.cpp 调用:

```
//file1.cpp

void fn();
void staticFn();    //link error

int main()
{
   fn();
   staticFn();
}

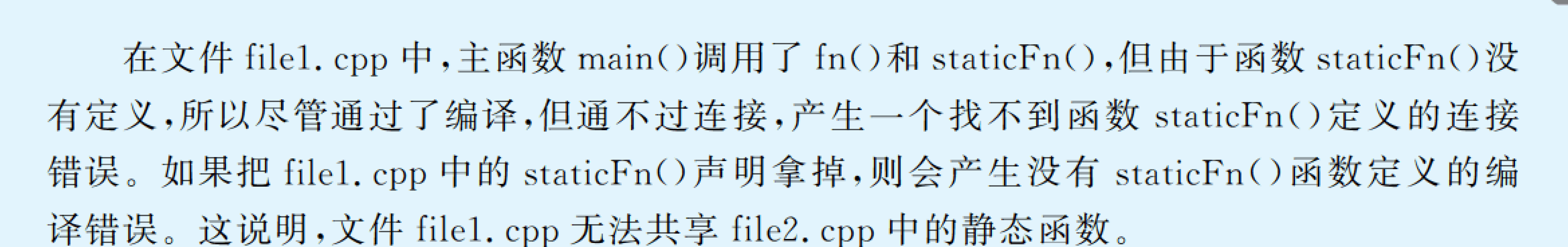
//file2.cpp

#include <iostream.h>

static void staticFn();
void fn();

void fn()
{
   staticFn();
   cout <<"this is fn()\n";
}

void staticFn()
{
   cout <<"this is staticFn()\n";
}
```



在文件 file1. cpp 中，主函数 main()调用了 fn()和 staticFn()，但由于函数 staticFn()没有定义，所以尽管通过了编译，但通不过连接，产生一个找不到函数 staticFn()定义的连接错误。如果把 file1. cpp 中的 staticFn()声明拿掉，则会产生没有 staticFn()函数定义的编译错误。这说明，文件 file1. cpp 无法共享 file2. cpp 中的静态函数。

在文件 file1. cpp 中，去掉关于函数 staticFn()的声明和调用，便能通过编译和连接了。

例如，下面的程序修改了上面代码的内容，其工程文件为 ch6_3. prj。其工程文件和源文件内容分别为：

```
// **********************
// ch6_3.prj
// **********************
ch6_3.cpp
ch6_3_1.cpp
// **********************

//----------------------
//     ch6_3.cpp
//----------------------
void fn();
//----------------------
int main(){
  fn();
}//--------------------

//----------------------
//     ch6_3_1.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
static void staticFn();
void fn();
//----------------------
void fn(){
  staticFn();
  cout <<"this is fn()\n";
}//--------------------
void staticFn(){
  cout <<"this is staticFn()\n";
}//--------------------
```

运行结果为：

```
c:\>ch6_3
this is staticFn()
this is fn()
```

主函数 main()调用了函数 fn()，fn()在文件 file2. cpp 中定义，所以它在 file2. cpp 中有效，于是可以随意调用该文件的另一个函数 staticFn()。

文件 file1. cpp 由于没有使用输出语句“cout << …”，所以不用包含头文件 iostream。静态有两个效果：第一，它允许其他源文件建立并使用同名的函数，而不相互冲突，在大的



编程项目中它是一个优势;第二,声明为静态的函数不能被其他源文件所调用,因为它的名字不能得到。假设要编写逻辑上只能由函数 A 调用的函数 B,其他函数对函数 B 的调用是无意义的,但是要求其他源文件能调用函数 A,把函数 B 声明为静态即能实现。

在文件作用域下声明的 inline 函数默认为 static 存储类型。在文件作用域下声明的 const 的常量也默认为 static 存储类型。它们如果加上 extern,则为外部存储类型。

6.3 作用域

作用域是标识符在程序中有效的范围,标识符的引入与声明有关,作用域开始于标识符的声明处。C++的作用域范围分为局部作用域(块作用域)、函数作用域、函数原型作用域、文件作用域和类作用域(见 11.7 节)。

1. 局部作用域

当标识符的声明出现在由一对大括号所括起来的一段程序(块)内时,该标识符的作用域从声明点开始,到块结束处为止。作用域的范围具有局部性。

例如,下面的代码描述了局部作用域:

```
#include <iostream>
using namespace std;
void fn(int y)        //y 的作用域从此开始
{
   int x = 1;         //x 的作用域从此开始
   if(x > y)
     cout << x << endl;
   else
     cout << y << endl;
   //...
}                     //x 和 y 的作用域到此结束
```

语句是一个程序单位,如果在 if 语句和 switch 语句中进行条件测试的表达式中声明标识符,则该标识符的作用域在该语句内。

例如,下面的代码在 if 语句中声明了一个局部作用域的标识符:

```
#include <iostream>
using namespace std;
void fn()
{
   if(int i = 5)         //i 的作用域从此开始(条件表达式可以是变量定义)
     i = 2 * i;
   else
     i = 100;
                         //i 的作用域到此结束
   cout << i << endl;    //error i 无定义
}
```

又如,下面的代码在 switch 语句中声明了一个块作用域的标识符:

```
#include <iostream>
using namespace std;
void fn()
{
   switch(int i = f())        //i 作用域从此开始(switch 中的整数表达式可以是变量定义)
   {
         case 1:
            cout << i << endl;
         //...
   }                          //i 作用域到此结束
   cout << i << endl;         //error i 无定义
}
```

在 if 条件表达式判断之后的语句和 else 之后的语句为一个语句块,尽管有时仅仅只是一条语句。

例如,下面的代码错用了 if 语句中声明的变量:

```
#include <iostream>
using namespace std;
void fn(int n)
{
   if(n>5)
        int i = n;            //整型 i 的作用域从此开始,到此结束
   else
        double i = n;         //double i 的作用域从此开始,到此结束

   cout << i << endl;         //error i 无定义
}
```

在 for 语句的第一个表达式中声明的标识符,其作用域在该语句内。

```
#include <iostream>
using namespace std;
void fn()
{
   int number, i;                      //i 的作用域从此开始
   for(i = 0; i<10; i++)
   {
        int halfNumber;
        if(i%2)
          number + = 1;
   }

   halfNumber = number/2;              //error halfNumber 无定义
   number = i;                         //ok
}
```

→ 在 C++早些时候的版本,例如 TC++ 3.0,允许 for 语句头中声明的变量的作用域延伸至包含 for 的最小块结束。后来的 ANSI C++标准规定,for 语句头中声明的变量的作用域只在该 for 语句中。但有些编译器仍保留其另一种意义的选择。如 BORLAND C++ 5.0 在 Options|Project|C++Options|C++Compatibility 中,可以选择 for 语句头中声明的变量的作用域范围。



2. 函数作用域

标号是唯一具有函数作用域的标识符。goto 语句使用标号。标号声明使得该标识符在一个函数内的任何位置均可以被使用。

例如,下面的代码声明了两个标号:

```
#include <iostream>
using namespace std;
void fn()
{
   goto S;
   int b;
   cin >> b;
   if(b > 0)
   {
S:
        goto End;
   }
End:
   Cout <<"All right\n";
}
```

goto 或 switch 语句不应使控制从一个声明的作用域之外跳到该声明的作用域内,因为这种跳转越过了变量的声明语句,使变量不能被初始化。

例如,下面的代码在 switch 语句内作了一个永远也到不了的变量说明:

```
switch(s)
{
   int b = 5;       //warning 无法到达此处
   case 1:
     b++;           //error: 变量 b 无定义
   //...
}
```

局部变量不具有函数作用域。

例如,下面的代码描述一个局部变量 k 在声明之前错误地进行了使用:

```
void fn()
{
   int n = 1;
   n = k + 1;       //error: 因为 k 不是函数作用域,只能从定义处开始有效
   int k;           //k 为块作用域
   //...
}
```

3. 函数原型作用域

函数原型声明(不是函数定义)中所作的参数声明在该作用域中。这个作用域开始于函数原型声明的左括号,结束于函数原型声明的右括号。

例如,下面的代码是函数 Area()的原型声明:

```
void Area(double width, double length);
```

参数声明“double width, double length”只在括号之内有效,在程序的其他地方使用 width 和 length 必须另外有定义,否则会引起无定义的标识符错。

例如,下面的代码引起一个无定义的标识符编译错误:

```
void Area(double width,double length);

length = 50;     //error length 无定义
```

所以,在这个函数原型声明中的标识符 width 和 length 是可有可无的。即上面的函数原型等价于下面的函数原型声明:

```
void Area(double, double);
```

但是,参数中有了标识符可以增强可读性。上面参数中带标识符的函数原型声明使人一看就明白一个参数是宽度值,另一个参数是长度值。所以,习惯上,在函数原型声明中,都为参数指定一个有说明意义的标识符,而且一般总是与该函数定义中参数的标识符一致。即:

```
void fn(int number);        //函数声明

//...

void fn(int number)         //函数定义
{
   //...
}
```

4. 文件作用域

文件作用域是在所有函数定义之外说明的,其作用域从说明点开始,一直延伸到源文件结束。

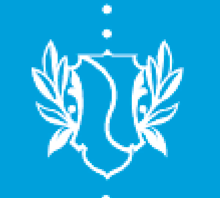
例如,下面的代码声明了 3 个全局变量,但由于声明点的差异,使得在函数中出现了编译错误:

```
//abc.cpp

int number;
static int value;

int main()
{
   a = 1;        //error a 无定义
   number = 0;
   value = number + 1;
}
int a;           //定义全局变量 a
```

value 是静态全局变量,它是文件作用域的。



静态全局变量是文件作用域的,静态函数也是文件作用域的。所以,文件作用域即为静态全局的。

在头文件的文件作用域中所进行的声明,若该头文件被一个源文件嵌入,则声明的作用域也扩展到该源文件中,直到源文件结束。

例如,cout 和 cin 都是在头文件 iostream.h 的文件作用域中声明的标识符,这两个标识符的作用域延伸到嵌入 iostream.h 的源文件中。

6.4 可见性

可见性从另一角度表现标识符的有效性,标识符在某个位置可见,表示该标识符可以被引用。可见性与作用域是一致的。作用域指的是标识符有效的范围,而可见性是分析在某一位置标识符的有效性。

可见性在分析两个同名标识符作用域嵌套的特殊情况时,非常有用。在内层作用域中,外层作用域中声明的同名标识符是不可见的,当在内层作用域中引用这个标识符时,表示的是对内层作用域中声明的标识符的引用。

例如,下面的程序定义 3 个不同作用域的同名变量,在访问它们时,可见性起了作用:

```
//----------------------
//    ch6_4.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
int id=3;
//----------------------
int main(){
  int id=5;
  {
    int id;
    id=7;
    cout<<"id="<<id<<"\n";
  }
  cout<<"id="<<id<<"\n";
}//---------------------
```

运行结果为:

```
id=7
id=5
```

这里外层初始化为 3 的变量 id 有文件作用域,内层初始化为 5 的变量 id 和最内层未初始化的变量 id 有块作用域。内层的变量 id 会隐藏外层的变量 id,因此,在主函数 main()内定义了变量 id 之后,就不能简单访问文件作用域的变量 id 了。

在最内层的块作用域结束后,次外层的块作用域又变得可见了,所以运行结果的第二行输出 id 的值为 5。

标识符的可见性范围不超过作用域,作用域则包含可见范围。

例如，下面的代码进一步说明可见性与作用域的关系：

```
{
   int i;
   char ch;
   i = 3;
   {
     double i;
     i = 3.0e3;            //int i 被隐藏
     ch = 'A';             //char ch 仍可见
   }                       //double i 的作用域结束
   i + = 1;                //int i 可见
}                          //int i,char ch 作用域结束
```

该代码的图示见图6-1。

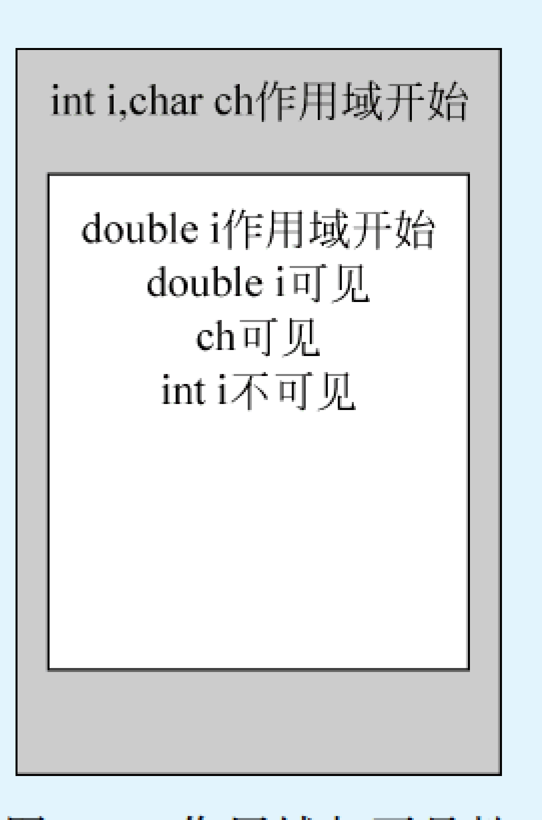


图6-1 作用域与可见性

图中阴影部分为 int i 和 char ch 的作用域。进入内部块后，double i 遮住了 int i，使得 int i 不可见，但变量 ch 仍可见。所以在内部块中，double i 的作用域和可见性是一致的，int i 的作用域存在，但不可见。在外部块中，ch 的作用域与可见性是一致的，因为 ch 的可见性渗透至内部块中。

如果被隐藏的是全局变量，则可用符号::来引用该全局变量。相关的内容可参见11.3节。

例如，下面的代码定义了同名的全局变量和局部变量，但仍可在局部作用域中访问全局变量：

```
int s = 0;

void f()
{
   float s = 3.0;
   int a;
   {
     float a = 2.0;
     ::a = 1;     //error,没有全局变量 a
     ::s = 1;     //ok,全局变量 s
     s = 2.0;     //ok,指 float s
   }
}
```



6.5 生命期

生命期也叫生存期。生命期与存储区域密切相关,存储区域主要有代码区(code area)、数据区(data area)、栈区(stack area)和堆区(heap area),见图5-2,对应的生命期为静态生命期、局部生命期和动态生命期。

1. 静态生命期

这种生命期与程序的运行期相同,只要程序一开始运行,这种生命期的变量就存在;当程序结束时,其生命期就结束。变量在固定的数据区中分配空间的,具有静态生命期。所以,全局变量、静态全局变量、静态局部变量都具有静态生命期。**具有文件作用域的变量具有静态生命期**。

例如,下面的代码声明了一个全局变量和一个局部变量,由于生命期不同,其命运也不同:

```
#include <iostream>
using namespace std;
int n;

int main()
{
    double m = 3.8;
    cout <<"n = " << n << endl;      //ok
    fn();
}

void fn()
{
    cout <<"m = " << m << endl;      //error
    cout <<"n = " << n << endl;      //ok
  }
```

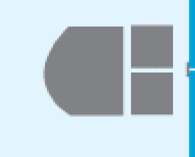
因为n是全局变量,所以任何函数都能够访问它。而m是局部变量,尽管它在调用函数fn()之前有了定义,但fn()仍无法访问它,原因是m不具有静态生命期,在fn()中不存在。**被调用函数不能享有使用调用函数中数据的权利**。

静态生命期的变量,若无显式初始化,则自动初始化为0。

函数驻在代码区,也具有静态生命期。在函数内部可以声明静态生命期的变量,即静态局部变量。

2. 局部生命期

在函数内部声明的变量或者是在块中声明的变量具有局部生命期。这种变量的生命期开始于程序执行经过其声明点时,而结束于其作用域结束处。所以具有局部生命期的变量也具有局部作用域。但反之不然,具有局部作用域的变量若为局部变量,则具有局部生命期;若为静态局部变量,则具有静态生命期。静态局部变量的生命期是从定义它的函数第一次被调用时开始存在,直到程序运行结束。



具有局部生命期的变量驻在内存的栈区。

具有局部生命期的变量如果未被初始化，则内容不可知。

3. 动态生命期

这种生命期由程序中特定的函数调用(malloc()和 free())或操作符(new 和 delete)来创建和释放，见 8.4 节。

具有这种生命期的变量驻在内存的堆中。当用函数 malloc()或 new 为变量分配空间时，生命期开始；当用 free()或 delete 释放该变量的空间或程序结束时，生命期结束。

6.6 头文件

一个程序经常由多个源文件组成，每个源文件是一个可编译的程序单位。在将程序分解成多个源文件之后，必须计划在每个源文件中哪些信息可以被其他文件可见，哪些不可见。C++提供了开放和隐藏信息的工具，那就是使变量或函数具有外部或静态存储类型或都不具有。

在程序中可能有：

- 全局变量声明，如 extern int n；
- 全局变量定义，如 int n；
- 静态全局变量定义，如 static int n；
- 静态函数声明，如 static void fn()；
- 函数声明，如 void fn()；
- 函数定义，如 void fn(){ //… }
- 类型声明，如 enum COLOR{ //… }；
- 全局常量声明，如 extern const float pi；
- 全局常量定义，如 const float pi=3.14；
- 内联函数定义，如 inline void fn()；
- 非外部或静态存储类型名字的声明及定义。

同一名字的声明可以多次，具有外部存储类型的声明可以在多个源文件中引用，因此方便的方法是将它们放在头文件中。头文件起着源文件之间接口的作用。

头文件一般可包含：

- 类型声明，如 enum COLOR{ //… }
- 函数声明，如 extern int fn(char s)；
- 内联函数定义，如 inline char fn(char p){ return *p++； }
- 常量定义，如 const float pi=3.14；
- 数据声明，如 extern int m； extern int a[]；
- 枚举，如 enum BOOLEAN{ false， true}；
- 包含指令(可嵌套)，如#include <iostream.h>
- 宏定义，如#define Case break;case
- 注释，如//check for end of file



这些经验规则说明哪些可以放在头文件中，哪些不可以放在头文件中。不是语言要这么做，而是对#include机制使用方法的一个合理建议。

但头文件不宜于包含：

- 一般函数定义，如 char fn(char p){return *p++;}
- 数据定义，如 int a; int b[5];
- 常量聚集定义，如 const int c[]={1,2,3};

例如，下面的程序由一个工程文件 ch6_5.prj 定义。该工程文件由 3 个源文件组成：

```
//*********************
//    ch6_5.prj
//*********************
ch6_5.cpp
mycircle.cpp
myrect.cpp
//****************** ***
```

这 3 个源文件中都包含了自己定义的头文件 myarea.h。这 3 个源文件分别为调用不同功能计算面积、定义计算圆面积的函数和定义计算矩形面积的函数。

```
//---------------------
//    myarea.h
//---------------------
double circle(double radius);
double rect(double width, double length);


//---------------------
//    mycircle.cpp
//    计算圆面积
//---------------------
#include"myarea.h"
//---------------------
const float pi = 3.14;
//---------------------
double circle(double radius){
  return pi * radius * radius;
}//--------------------

//---------------------
//    myrect.cpp
//    计算矩形面积
//---------------------
#include"myarea.h"
//---------------------
double rect(double width,double length){
  return width * length;
}//--------------------

//---------------------
//    ch6_5.cpp
//    主函数
//---------------------
#include"myarea.h"
#include<iostream>
```

```
using namespace std;
//----------------------
int main(){
  double width,length;
  cout<<"please enter two numbers:\n";
  cin>>width>>length;

  cout<<"area of rectangle is "<<rect(width,length)<<endl;

  double radius;
  cout<<"please enter a radius:\n";
  cin>>radius;

  cout<<"area of circle is "<<circle(radius)<<endl;
```

运行结果为：

```
c:\>ch6_5
please enter two numbers:
5.5  2.0
area of rectangle is 11
please enter a radius:
2.1
area of circle is 13.8474
```

要开发该程序，首先分别编辑头文件 myarea.h 和其他 C++源文件。

由于 ch6_5.cpp 中用到了流输出，所以必须包含 C++I/O 流的头文件 iostream。

对工程文件进行编译和连接，产生一个名为 ch6_5.exe 的执行程序，运行之，就会得到应有的结果。

工程文件中的 3 个文件都含有 myarea.h 的头文件，这样可以使信息共用，保证程序的一致，在大型程序开发中尤为必要。

6.7 多文件结构

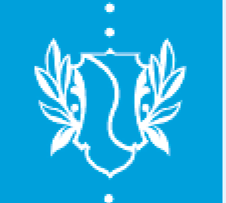
程序开发的示意图见图 6-2。

图中，源文件中含有包含头文件的预编译语句，经过预编译后，产生翻译单元，该翻译单元以临时文件的形式存放在计算机中。之后编译，进行语法检查，产生目标文件(.obj)。若干个目标文件经过连接，产生可执行文件(.exe)。连接包括 C++库函数的连接和标准类库的连接。

许多小程序可以由单个源文件建立，它编译成一个目标文件(.obj)，然后输给连接器，产生运行程序。这样的程序维护方便。如果修改了源文件中的任何函数，只需再次启动编译器。

大程序倾向于分成多个源文件，其理由为：

(1) 避免一而再、再而三地重复编译函数。因为编译器总是以文件为单位工作的，如果一个文件中包含的函数太多，则由于被修改的函数总是少数的几个，所以大多数正确的函数都得重新编译一次。



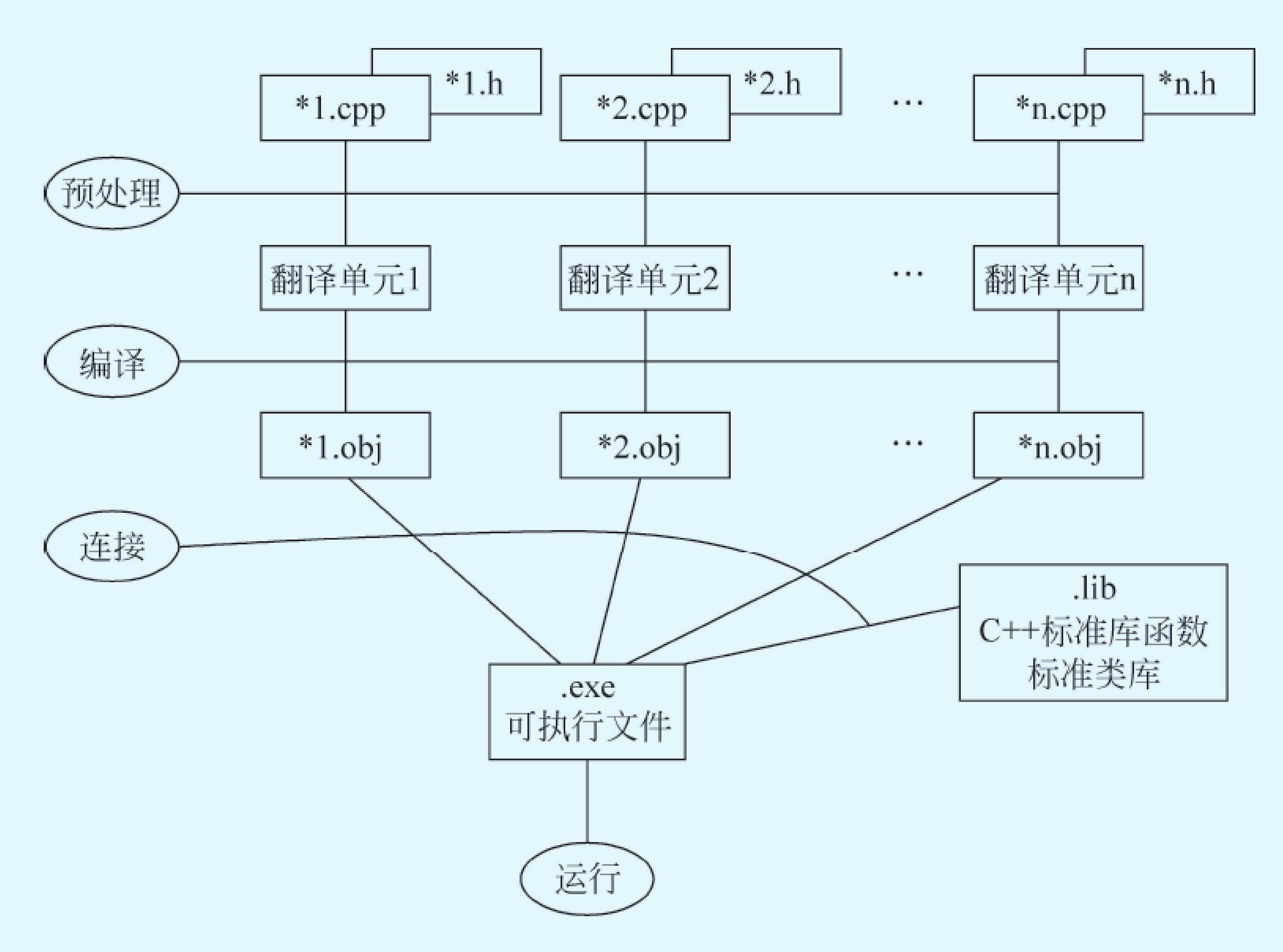


图 6-2　C++程序开发示意图

(2) 使程序更加容易管理。可以将程序按逻辑功能划分,分解成各个源文件,便于程序员的任务安排以及程序调试。

(3) 把相关函数放到一特定源文件中。例如,所有输入函数放在一个源文件中。

6.8　编译预处理

预处理程序也称预处理器,它包含在编译器中。预处理程序首先读源文件。预处理的输出是“翻译单元”,它是存放在内存中的临时文件。编译器接受预处理的输出,并把源代码转化成包含机器语言指令的目标文件。

预处理程序对源文件进行第一次处理,它处理的是预处理指令。我们介绍三类预处理指令:#include、#define 和 #if。

1. #include(包含)指令

include 命令让预处理器把一个源文件嵌入到当前源文件中该点处。它有两种格式,一种格式是:

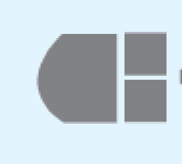
```
#include <文件名>
```

这种格式用于嵌入 C++提供的头文件。这些头文件一般存于 C++系统目录中的 include 子目录下。C++预处理器遇到这条指令后,就到 include 子目录下搜索给出的文件,并把它嵌入到当前文件中。这种方式是标准方式。

另一种格式是:

```
#include "文件名"
```

预处理器遇到这种格式的包含指令后,首先在当前文件所在目录中进行搜索,如果找不



到，再按标准方式进行搜索。这种方式适合于规定用户自己建立的头文件。

include 文件可以嵌套，即在头文件中还可以有包含指令。

2. ＃define(宏定义)指令

在 C 中，＃define 最常用的方法是建立常量，但已经被 C++的 const 定义语句所代替，见 2.5 节。

＃define 还可以定义带参数的宏，但也已经被 C++的 inline 内嵌函数所代替，见5.7 节。

＃define 的一个有效的使用是在条件编译指令中。

3. 条件编译指令

条件编译的指令有＃if、＃else、＃elif、＃endif、＃ifdef、＃ifndef 和＃undef。

条件编译的一个有效使用是协调多个头文件。

例如，符号 NULL 在 6 个不同的头文件中都有定义：locate. h、stddef. h、stdio. h、stdlib. h、string. h 和 time. h。一个源文件可能包含其中的几个头文件，这样会使得编译给出"一个符号重复定义多次"的错误。这时，需要在每个头文件中使用条件编译指令：

```
#ifndef NULL
   #define NULL ((void *)0)
#endif
```

上面的代码能够保证符号 NULL 在一个程序中只有一次定义((void *)0)。而当再次遇到头文件时，一切定义的企图都被＃ifndef 给"挡驾"了。

使用＃undef 可以取消符号定义，这样可以根据需要打开和关闭符号。

小结

存储类型决定了名字在内存中的位置，存储类型也规定了在多文件程序中的连接特性。

非静态的全局名字具有外部存储类型的属性，它使得程序中的诸文件之间共享该名字，前提是在程序的各文件中当且仅当只有一个名字定义而其他皆为声明。

静态就是让变量和函数在声明的区域内成为私有。这使得多文件协作编程的数据交错使用状态下可靠性增强。

作用域规则规定了程序中名字的有效范围，它给名字的可见性提供了依据。所有的变量都有作用域和可见性。

为使在不同源文件中保持声明的一致性，采用头文件。

预处理器对源文件进行初次处理，处理时，它忽略注释语句，加入. h 头文件，并按宏定义进行替换。

在面向对象程序设计中，程序结构的意义扩展到了类。类作用域、名空间、类及对象的连接特性以及程序的合理分解都是新的内容，这些内容将在 11.7 节中继续介绍。



练习

6.1 指出下列程序的错误。

(1)

```
//file1.cpp

int x = 1;
int func()
{
   //...
}

//file2.cpp

extern int x;
int func();
void g()
{
   x = func();
}

//file3.cpp

extern int x = 2;
int g();

int main()
{
   x = g();
   //...
}
```

(2)

```
//file1.cpp

int x = 5;
int y = 8;

extern int z;

//file2.cpp

int x;

extern double y;
extern int z;
```



6.2 写出下列程序的运行结果。

```
//file1.cpp

static int i = 20;
int x;

void f(int v)
{
   x = g(v);
}

static int g(int p)
{
   return i + p;
}

//file2.cpp

#include <iostream>
using namespace std;
extern int x;
void f(int);

int main()
{
   int i = 5;
   f(i);
   cout << x;
}
```

6.3 将习题 4.9 分解为 4 个源文件实现。一个文件含有主函数，调用其他 3 个函数。其他 3 个文件分别含有一个乘法九九表输出格式的函数定义。要求用一个头文件作为相互联络的接口。

第7章 数组

前面介绍的数据都是基本数据类型(整型、字符型、浮点型)。人们经常需要使用大量集中在一起的数据来工作,C++支持数组处理来满足这一需求。数组可以是一维的,也可以是多维的,许多重要的应用都是基于数组的。学习本章后,要求理解数组下标,掌握初始化数组的方法,学会把数组用作函数参数,学会二维数组的使用,并学习数组应用的技术。

7.1 数组的概念

数组是一个由若干同类型变量组成的集合。一维数组的说明方法为数据类型加数组名,再加方括号,里面含有元素个数。即:

```
类型说明符 数组名[常量表达式];
```

例如,下面的代码说明一个字符数组:

```
int main()
{
   char buffer[5];
   //...
}
```

主函数中定义了一个字符数组,该数组占5个字节。这个字符数组可以是最长为4个字符的单词,因为第5个字节用于'\0'字符,用'\0'字符结束的字符数组构成一个字符串。

数组名的命名规则和变量名是一样的。数组名后是用方括号括起来的常量表达式,不能用圆括号。例如,下面的用法不对:

```
int a(5);                //并非数组,而是初值为5的变量a定义
```

常量表达式表示元素的个数,即数组长度。“char a[5]”表示a数组有5个元素,每个元素的类型是字符型。数组下标从0开始,分别是a[0],a[1],a[2],a[3],a[4]。注意,a[5]不属于该数组的空间范围。数组的内存排列见图7-1。

下标是数组元素到数组开始的偏移量。第 1 个元素的偏移量是 0，第 2 个元素的偏移量是 1，以此类推。由此，数组是一系列大小相同的连续项，每项到公共基点的偏移量是固定的。

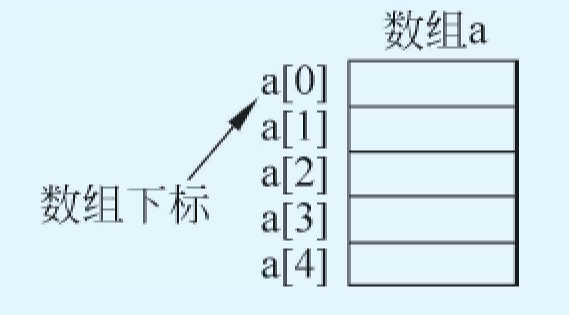


图 7-1　数组的内存排列

数组定义的方括号中，常量表达式可以包含枚举常量和字符常量，该常量表达式的值是在编译时确定的。一个数组定义是具有确定含义的操作，它分配固定大小的空间。如果方括号的值不能在编译时确定，那就只能在运行时确定，即在函数调用时即兴分配数组空间。这使得为局部作用域的数组分配数据空间的语句具有不同的意义，它随每次函数调用的不同而不同，这是不允许的。

C++允许堆内存分配来建立数组，8.4 节介绍了这种方法。

数组的作用域规则和单个变量相同。定义为局部作用域的数组，刚分配完空间时，其内容是不定的。全局作用域数组和静态局部作用域数组初始为全 0。例如，下面的代码用 3 种方法定义数组：

```
int iArray[10];                  //全局数组
void funcA();
void funcB();
int main()
{
   funcA();
   funcB();
}

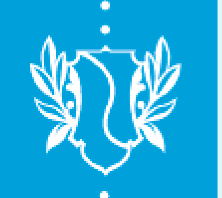
void funcA()
{
   static int iStaticLocal[30];  //局部静态数组
   //...
}

void funcB()
{
   int iLocal[20];               //局部数组
   //...
}
```

数组 iArray 在所有函数外面定义，它是全局的，因此，可以被任何函数访问。这个数组没有初始化，所以数组的每个元素都初始为 0。在 16 位机器上，这个数组中每个元素占 2 个字节，所以共占 20 个字节内存空间。

函数 funcA()定义了一个静态局部数组，它有 30 个整型数，这个数组不在栈中，而是在全局数据区。它没有显式初始化，所以默认初始值为 0，与全局数组相同。但在函数 funcB()中，iStaticLocal 是不可见的。因为是静态的，数组在 funcA()调用期间保持其值。如果在第一次调用 funcA()时改变了 iStaticLocal 的值，则第二次调用 funcA()时，funcA()拥有这些改变了的值，详见 5.5 节。

函数 funcB()定义了在栈中分配的数组，它有 20 个整型数。由于该数组在栈中，它受到了栈空间大小的限制。如果定义数组的元素很多(如 500 000)，则有可能使程序运行由于不能满足数组分配而突然终止。**不能满足内存分配要到程序运行时才知道**，因为**编译只管**



语法检查,而不管运行环境。程序连接时,才确定各内存空间包括栈空间的大小。确定栈空间大小后,程序运行中遇到大容量数组分配而不能满足时,才会有所表示。

编程时,如果要定义一个很大的数组,可以通过将其定义为静态或全局来解决,也可以将其在堆内存中分配(见8.4节)。

在编译时,数组定义中的下标必须确定。例如,下面的代码不能通过编译:

```
int main()
{
   int size = 50;
   int array[size];     //error: 不能用变量来描述数组定义中的元素个数
//...
}
```

尽管变量size已经赋有值50,紧接着就是数组的定义,阅读上都能理解,但是size是变量这一性质是编译不能原谅的。对于常量,编译可以用一个值直接代替,但对于变量,编译不能用值来代替,而是编译为取该变量的值。**取值是一个操作,不是值本身**,不能决定数组下标。程序运行中,通常通过常量来决定数组大小。

用全局变量的值来确定数组下标也是不允许的。例如:

```
int size = 50;

int main()
{
   int array[size];     //error
   //...
}
```

任何函数都在程序运行中被调用,函数被调用的先后次序是未知的。在函数被调用时,全局变量的值也是不可预测的,所以,数组的下标无法确定。

例如,下面的代码用常量来规定数组元素个数:

```
const int size = 50;
const int n = size * sizeof(int);

int main()
{
   int array[size];     //ok
   char charray[n];     //ok
   //...
}
```

size和n都是常量,n的常量定义中,初始化值是个表达式,但在编译时,该表达式的值能被确定下来。因此,编译总是认可数组定义中用常量说明的元素个数。

7.2 访问数组元素

数组中特定的元素通过下标访问。在C++中,所有数组均由连续的存储单元组成,起始地址对应于数组的第一个元素,下标是距数组开始的偏移量。长度为n的数组,其下标范围为0～(n－1)。



在内存的表示中，地址是从 0 开始的。如果表示数组的下标从 1 开始，则需要进行额外的机器操作，C 或 C++的处理方法使其编译器更简单有效，使代码效率更高。数组直接从 0 下标开始则不必进行这种多余的调整。

在数组定义后，给数组赋初值时，必须一个个元素逐个访问。例如，下面的代码给一个数组赋一组 Fibonacci 数：

```
int main()
{
   int iArray[10];    //数组定义

   iArray[0] = 1;
   iArray[1] = 1;
   iArray[2] = 2;
   iArray[3] = 3;
   //...
   iArray[9] = 55;
   //...
}
```

如果知道元素之间的规律，上面的赋值也可以通过循环来完成：

```
int main()
{
   int iArray[10];
   iArray[0] = 1;
   iArray[1] = 1;
   for(int i = 2; i < 10; i ++ )
     iArray[i] = iArray[i - 1] + iArray[i - 2];
   //...
}
```

for 循环的终止条件为 i<10。当 i==10 时，它终止循环。必须保证没有超出数组边界。如果循环条件误写成"i<=10"，那么程序将会执行到包括"iArray[10]=90;"的语句，改变不属于数组空间的内存单元(见图 7-2)。这个代码的失误不会在程序的编译与连接中反映出来，而是可能一直运行下去，直到出现结果不正确，或严重时导致死机。

	iArray 数组
数组下标 0	1
1	2
2	3
3	4
4	5
5	8
6	13
7	21
8	34
9	55
非数组元素	90

图 7-2　数组越界的内存表示

字符数组，实际上是 1 字节的整数数组。处理字符数组的方法与处理其他数组相同。字符数组若用来存储字符串，则要考虑字符串末尾的'\0'结束符。



例如,下面的程序输出一个字符串:

```
//----------------------
//    ch7_1.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  char chArray[30];
  cin.get(chArray,30);
  for(int i=0; chArray[i]!='\0'; i++)
    cout << chArray[i];
  cout << endl;
}//--------------------
```

其中,get()是输入流的成员函数(我们暂且模仿之,到面向对象程序设计中,再去讨论其实现机制)。使用它时,前面必须加"cin."。它输入一系列字符,直到输入流中出现结束符或所读字符个数已达到要求读的字符个数。它的原型为:

```
get(char* target,int count,char delimeter='\n');
```

其中,target 为存放一系列字符的空间地址,count 为限制最长的读取字符个数,delimeter 为规定的结束符。遇到此结束符时,尽管还没有到达读取字符最大个数,也还是结束读入过程。

这里 for 循环保持下去的条件为:

```
chArray[i]!='\0';
```

实际上没有必要让 i 达到 30。用户也许不会输入这么多字符。cin.get()在用户输入的最后字符后面加上'\0'。人们对'\0'之后的无定义字符没有兴趣。

可以用"chArray[i];"来代替前面的终止条件,也即 for 循环可以写成:

```
for(int i=0; chArray[i]; i++)
```

因为字符串中的所有字符,除最后的'\0'外,都是非 0(真值)值,所以,较短的格式是可行的。这两种表示在应用中都很普遍。

字符串的处理在 8.7 节中将进一步展开讨论,读者将会看到字符串的输出无须逐个字符输出,可以一种更简捷的方式进行。

7.3 初始化数组

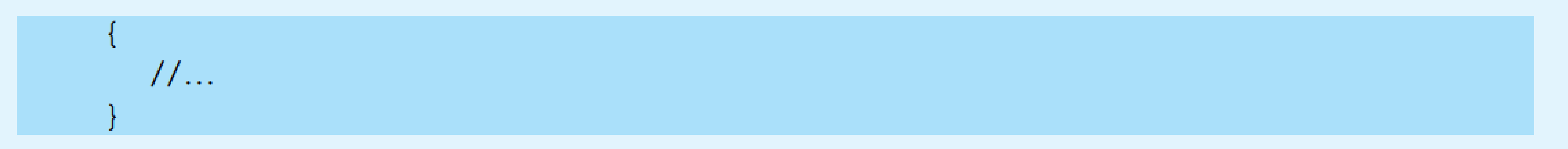
1. 数组的初始化

数组可以初始化,即在定义时,使它包含程序马上能使用的值。

例如,下面的代码定义了一个全局数组,并用一组 Fibonacci 数初始化:

```
int iArray[10]={1,1,2,3,5,8,13,21,34,55};     //初始化

int main()
```



```
{
   //...
}
```

初始化数组的值的个数不能多于数组元素个数，初始化数组的值也不能通过跳过逗号的方式来省略，这在 C 中是允许的，但在 C++ 中不允许。

例如，下面的代码对数组进行初始化是错误的：

```
int array1[5] = {1,2,3,4,5,6};      //error:初始化值个数多于数组元素个数
int array2[5] = {1,,2,3,4};         //error: 初始化值不能省略
int array3[5] = {1,2,3,};           //error: 初始化值不能省略
int array4[5] = {};                 //error: 语法格式错误

int main()
{
   //...
}
```

初始化值的个数可少于数组元素个数。当初始化值的个数少于数组元素个数时，前面的按序初始化相应值，后面的初始化为 0。

例如，下面的程序对数组进行初始化：

```
//----------------------
//    ch7_2.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int array1[5] = {1,2,3};
static int array2[5] = {1};
//----------------------
int main(){
  int arr1[5] = { 2 };
  static int arr2[5] = { 1, 2 };
  cout <<"global:\n";
  for(int n = 0; n < 5; n++)
    cout <<" "<< array1[n];

  cout <<"\nglobal static:\n";
  for(int n = 0; n < 5; n++)
    cout <<" "<< array2[n];

  cout <<"\nlocal:\n";
  for(int n = 0; n < 5; n++)
    cout <<" "<< arr1[n];

  cout <<"\nlocal static:\n";
  for(int n = 0; n < 5; n++)
    cout <<" "<< arr2[n];
  cout << endl;
}//--------------------
```

运行结果为：

```
global:
   1  2  3  0  0
global static:
   1  0  0  0  0
local:
   2  0  0  0  0
local static:
   1  2  0  0  0
```

其中，全局数组和全局静态数组的初始化是在主函数运行之前完成的，而局部数组和局部静态数组的初始化是在进入主函数后完成的。

全局数组 array1[5]对于初始化表的值按序初始化为 1，2，3，还有两个元素的值则按默认初始化为 0。

全局静态数组 array2[5]与全局数组的初始化情况一样，初始化表值{1}表示第 1 个元素的值，而不是指全部数组元素都为 1。

局部数组 arr1[5]根据初始化表值的内容按序初始化，初始化表值虽只有 1 个，但因启动了初始化，使得其余元素都被默认初始化为 0 了。

局部静态数组 arr2[5]先根据初始化表按序初始化，其余 3 个数组元素的值默认初始化为 0。

2. 初始化字符数组

初始化字符数组有两种方法，一种是：

```
char array[10] = {"hello"};
```

另一种是：

```
char array[10] = {'h','e','l','l','o','\0'};
```

第一种方法用途较广，初始化时，系统自动在数组没有填值的位置用'\0'补上。另外，这种方法中的大括号可以省略，即能表示成：

```
char array[10] = "hello";
```

第二种方法一次一个元素地初始化数组，如同初始化整型数组。这种方法通常用于输入不容易在键盘上生成的那些不可见字符。

例如，下面的代码中初始化值为若干制表符：

```
char chArray[5] = {'\t','\t','\t','\t','\0'};
```

这里不要忘记为最后的'\0'分配空间。如果要初始化一个字符串"hello"，那为它定义的数组至少有 6 个数组元素。

例如，下面的代码给数组初始化，但会引起不可预料的错误：

```
char array[5] = "hello";
```

该代码不会引起编译错误，但由于改写了数组空间以外的内存单元，所以是危险的。



3. 省略数组大小

有初始化的数组定义可以省略方括号中的数组大小。

例如，下面的代码中数组定义为 5 个元素：

```
int a[ ] = {2,4,6,8,10};
```

编译时必须知道数组的大小。通常，声明数组时方括号内的数字决定了数组的大小。有初始化的数组定义又省略方括号中的数组大小时，编译器统计大括号之间的元素个数，以求出数组的大小。

例如，下面的代码产生相同的数组空间：

```
static int a1[5] = {1,2,3,4,5};
static int a2[ ] = {1,2,3,4,5};
```

让编译器得出初始化数组的大小有几个好处。它常常用于初始化一个元素个数在初始化中确定的数组，提供程序员修改元素个数的机会。

在没有规定数组大小的情况下，怎么知道数组的大小呢？sizeof 操作解决了该问题。

例如，下面的代码用 sizeof 确定数组的大小：

```
//---------------------
//    ch7_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  static int a[ ] = {1,2,4,8,16};
  for(int i = 0; i<(sizeof(a)/sizeof(int)); i++)
    cout << a[i]<<" ";
  cout << endl;
}//--------------------
```

运行结果为：

```
1  2  4  8  16
```

sizeof 操作使 for 循环自动调整次数。如果要从初始化 a 数组的集合中增删元素，只需重新编译即可，其他内容无须改动。

每个数组所占的存储量都可以用 sizeof 操作来确定。sizeof 返回指定项的字节数。sizeof 常用于数组，使代码可在 16 位机器和 32 位机器之间移植。

对于字符串的初始化，要注意数组实际分配的空间大小是字符串中字符个数加上末尾的'\0'结束符。

例如，下面的代码定义一个字符数组：

```
//---------------------
//    ch7_4.cpp
//---------------------
#include <iostream>
#include <string.h>     //用到 strlen()
```




```
using namespace std;
//----------------------
int main(){
  char ch[] = "how are you";

  cout <<"size of array: "<< sizeof(ch)<< endl;
  cout <<"size of string: "<< strlen(ch)<< endl;
}//---------------------
```

运行结果为：

```
size of array: 12
size of string: 11
```

其中，数组大小为12，而字符串长度为11。

省略数组大小只能在有初始化的数组定义中。

例如，下面的代码将产生一个编译错误：

```
int a[]; //error: 没有确定数组大小
```

在定义数组的场合，无论如何，编译器必须知道数组的大小。

7.4 向函数传递数组

无论何时，将数组作为参数传给函数，实际上只是把数组的地址传给函数。

物理上，把整个数组放在栈中是不合理的，因为栈大小是一定且有限的。如果把传送给函数的整个数组都放在栈中(内存的大块复制)，则很快会把栈空间用光。

1. 传递给标准库函数

C++中有一个memset()函数，它可以一字节一字节地把整个内存区块设置为一个指定的值。memset()函数在string.h头文件中声明，它的第一个参数是内存区块的起始地址，第二个参数是设置每个字节的值，第三个参数是内存区块的长度(字节数，不是元素个数)。其函数原型为：

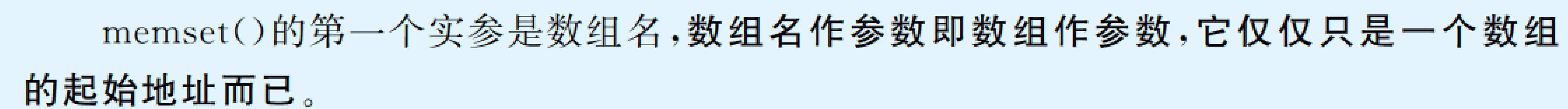
```
void * memset(void * , int, unsigned);
```

其中，void * 表示地址，详细介绍见8.6节。

例如，下面的代码用数组做参数传递给标准函数memset()，以让其将数组设置成全0：

```
#include <mem.h>

int main()
{
   int ia1[50];
   int ia2[500];
   memset(ia1, 0, 50 * sizeof(int));
   memset(ia2, 0, 500 * sizeof(int));
   //...
}
```


memset()的第一个实参是数组名，**数组名作参数即数组作参数，它仅仅只是一个数组的起始地址而已**。

实现第一个 memset()函数调用的内存布局见图 7-3。在函数 memset()栈区，从返回地址往上依次为第 1、2、3 个参数。第 1 个参数中的内容是 main()函数中定义的数组 ia1 的起始地址；第 2 个参数是给数组设置的值(0)；第 3 个参数是数组的长度(50×2)。函数返回时，main()函数的数组中内容全置为 0。

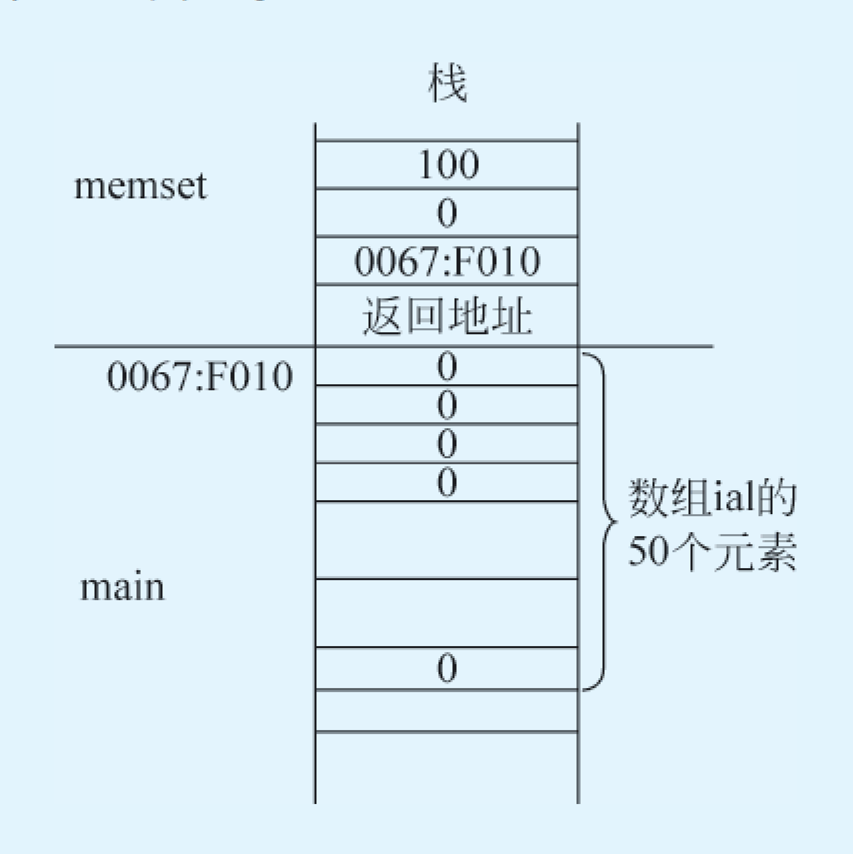


图 7-3　memset 函数调用的内存布局

2. 传递给自定义函数

若要让一个函数求数组元素的和，需传递一个数组参数和数组大小参数。数组大小参数是需要的，因为从传递的数组参数(地址)中，没有数组大小的信息。

例如，下面的程序调用一个函数求数组元素之和：

```
//----------------------
//    ch7_5.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int sum(int[], int);
//----------------------
int main(){
  static int ia[5] = {2,3,6,8,10};
  int sumOfArray = sum(ia, 5);
  cout << "sum of array: " << sumOfArray << endl;
}//---------------------
int sum(int array[], int len){
  int iSum = 0;
  for(int i = 0; i < len; i++)
    iSum += array[i];
  return iSum;
}//---------------------
```



运行结果为：

```
sum of array: 29
```

sum()函数以整数数组作为第一个参数，以整数作为第二个参数。由于传递数组实际上传递的是地址，所以函数原型中，数组参数的书写形式无须在方括号中写明数组大小。如果写明了数组大小，编译器将忽略之。数组形参的空方括号只是告诉函数，该参数是个数组的起始地址。由于数组参数是地址，对数组参数不能通过 sizeof 求得数组大小，所以 sum()函数必须要有第二个参数：数组的大小，即数组的元素个数。

7.5 二维数组

1. 二维数组定义

C++中的数组可以有多个下标，需要两个下标才能标识某个元素的数组称为二维数组。二维数组经常用来表示按行和列格式存放信息的数值表。要识别表中某个特定的元素，必须指定两个下标。习惯上，第一个下标表示该元素所在行，第二个下标表示该元素所在列。

图 7-4 中表示了一个名为 a 的 3 行×4 列的整型二维数组。可以看到，第一个下标范围是 0～2，第二个下标范围是 0～3。二维数组是按先行后列的顺序在内存中线性排列的。它的定义如下：

```
int a[3][4];
```

通常把有 m 行和 n 列的数组称为 m×n 数组。

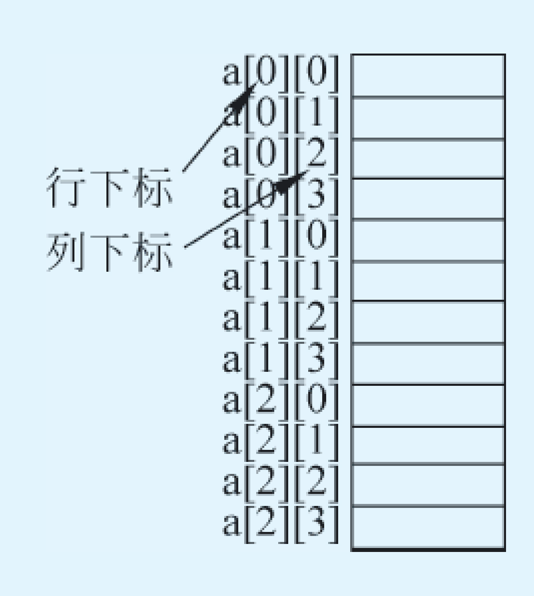


图 7-4　3×4 数组排列的内存表示

数组 a 中的每个元素用元素名 a[i][j]识别，其中，a 是数组名，i 和 j 是标识数组 a 中每个元素的下标。

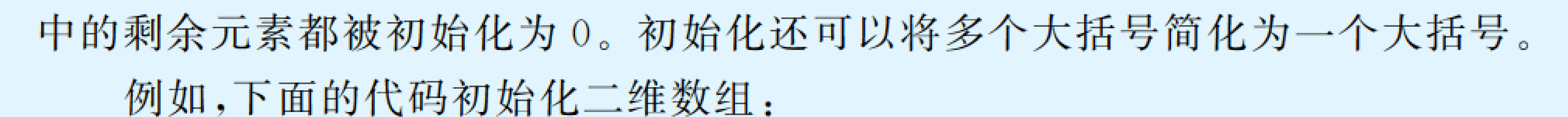
2. 初始化

和一维数组一样，二维数组也能在定义时被初始化。

例如，下面定义一个 2×3 的整型数组，并初始化：

```
int b[2][3] = {{1,2,3},{4,5,6}};
```

其中的值是按行用大括号组合在一起的。{1,2,3}初始化了 b[0][0]、b[0][1]和 b[0][2]，{4,5,6}初始化了 b[1][0]、b[1][1]和 b[1][2]。如果某行没有足够的初始化值，那么该行



中的剩余元素都被初始化为0。初始化还可以将多个大括号简化为一个大括号。

例如，下面的代码初始化二维数组：

```
//----------------------
//    ch7_6.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  int array1[2][3] = {1,2,3,4,5};
  int array2[2][3] = {{1,2},{4}};

  for(int i = 0; i<2; i++){     //按行列输出数组 array1
    for(int j = 0; j<3; j++)
      cout << array1[i][j]<<",";
    cout << endl;
  }
  cout << endl;
  for(int i = 0; i<2; i++){     //按行列输出数组 array2
    for(int j = 0; j<3; j++)
      cout << array2[i][j]<<",";
    cout << endl;
  }
}//--------------------
```

运行结果为：

```
1,2,3,
4,5,0,

1,2,0,
4,0,0,
```

数组array1提供了5个初始化值，这些初始化值先赋给第一行元素，然后再赋给第二行元素。最后一个元素array1[1][2]没有被明确初始化，在这里被默认初始化为0。数组array2的定义语句在两个大括号中提供了3个初始化值。用于第一行的初始化值表把第一行的前两个元素明确地初始化为1和2，第三个元素即默认为0。用于第二行的初始化值表把第二行的第一个元素明确地初始化为4，其余为0。

3. 省略第一维大小

如果对全部元素赋初值，则定义数组时对第一维的大小可以忽略，但第二维的大小不能省。例如：

```
int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

与下面的代码是等价的：

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

编译器会根据数据总个数分配空间，每行4列，所以确定该数组为3行。

在定义时，也可以只对部分元素赋初值而省略第一维的大小，但应分行赋初值。例如：




```
int a[][4] = {{1,2,3},{0},{4,5}};
```

该数组定义表示 3×4 的整型数组，没有明确初始化值的都默认初始化为 0，所以等价于下面的定义：

```
static int a[3][4] = {{1,2,3},{0,0,0},{4,5,0}};
```

访问二维数组中所有的元素通常需要一个二重循环。例如，上例(ch7_6.cpp)输出数组 array1 和 array2 所有元素的值。

4. 作为参数传递

作为参数传递一个二维数组给函数，其意义也为内存地址，所以原型中，声明整数数组参数的形式只能省略左边的方括号。

例如，下面的程序定义一个 3×4 的数组，表示 3 个学生，每个学生有 4 次测验成绩，求所有学生中的最好成绩。问题化作遍历二维数组找最大值，传递函数参数时，除了传递数组名外，还要传递行数和列数：

```
//----------------------
//    ch7_7.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
int maximum(int[][4],int,int);
//----------------------
int main(){
  int sg[3][4] = {{68,77,73,86},
                  {87,96,78,89},
                  {90,70,81,86}};
  cout<<"the max grade is "<<maximum(sg,3,4)<<endl;
}//---------------------
int maximum(int grade[][4], int pupils, int tests){
  int max = 0;
  for(int i = 0; i<pupils; i++)
    for(int j = 0; j<tests; j++)
      if(grade[i][j]>max)
        max = grade[i][j];

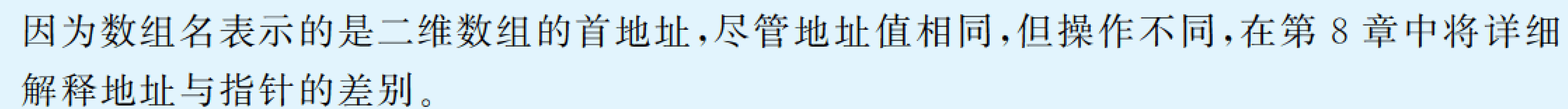
  return max;
}//---------------------
```

运行结果为：

```
the max grade is 96
```

5. 降维处理

由于二维数组在内存中是线性排列的，传递一维数组和传递二维数组都是传的地址，所以可以在被调用的函数中用单重循环来遍历二维数组中的所有元素，此时只需传递数组名和元素总个数。要注意被传递的数组地址不要用数组名表示，要用第一个元素的地址表示，



因为数组名表示的是二维数组的首地址，尽管地址值相同，但操作不同，在第 8 章中将详细解释地址与指针的差别。

例如，下面的程序是上例程序的改进。主函数将第一个数组元素地址作为一维数组首地址传递给 maximum()函数（实参），maximum()函数也以一维整型数组的首地址来接受，求取学生成绩的最大值：

```
//----------------------
//    ch7_8.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int maximum(int[],int);
//----------------------
int main(){
  int sg[3][4] = {{68,77,73,86},
                  {87,96,78,89},
                  {90,70,81,86}};
  cout <<"the max grade is "
       << maximum(&sg[0][0],3 * 4)   //传递第一个元素地址和元素个数
       << endl;
}//---------------------
int maximum(int grade[], int num){
  int max = 0;
  for(int i = 0; i < num; i++)
    if(grade[i]> max)
    max = grade[i];

  return max;
}//---------------------
```

运行结果为：

```
the max grade is 96
```

函数调用时，**数组参数的实参为整型变量的地址，函数原型中，数组参数的形参为整型数组的首地址**，二者类型是匹配的。进一步的讨论见 8.3 节和 8.8 节。

7.6 数组应用：排序

数据排序是重要的计算应用之一。排序方法有很多，先介绍最简单的排序法：冒泡排序法（bubble sort），然后介绍最常用的排序法：插入排序法（insert sort），最后介绍最快的排序法：快速排序法（quick sort）。

1. 冒泡排序法（bubble sort）

冒泡排序法可以形象地描述为：使较小的值像空气泡一样逐渐“上浮”到数组的顶部，或者较大的值逐渐“下沉”到数组的底部。这种排序技术要排序好几轮，每一轮都要比较连续的数组元素对。如果某对数值是按升序排列的，那就保持原样；如果按降序排列，就交换它们的值。冒泡排序法见图 7-5。

原数据	第一轮	第二轮	第三轮	第四轮
B	B	B	B	A
E	D	C	A	B
D	C	A	C	C
C	A	D	D	D
A	E	E	E	E

图 7-5　冒泡排序法

图 7-5 是将原数据序列 BEDCA 排序。第一轮结束时 E 沉底,第二轮结束时 D 下沉,第三轮结束时 C 下沉,第四轮结束时 B 下沉,从而得到从小到大的顺序排列。

例如,下面的程序把有 10 个元素的数组用冒泡排序法按升序排列:

```
//----------------------
//    ch7_9.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
void bubble(int[], int);
//----------------------
int main(){
  int array[] = {55,2,6,4,32,12,9,73,26,37};
  int len = sizeof(array)/sizeof(int);          //计算元素个数
  for(int i = 0; i < len; i++)                  //原始顺序输出
    cout << array[i]<<",";
  cout <<"\n\n";

  bubble(array, len);                           //调用自定义排序函数
}//----------------------
void bubble(int a[], int size){                 //冒泡排序
  for(int pass = 1; pass < size; pass++){       //共比较 size-1 轮
    for(int i = 0; i < size - pass; i++)        //比较一轮
      if(a[i]> a[i+1]){
        int temp = a[i];                        //交换元素
        a[i] = a[i+1];
        a[i+1] = temp;
      }
    for(int i = 0; i < size; i++)               //比较一轮后输出
      cout << a[i]<<",";
    cout << endl;
  }
}//----------------------
```

运行结果为:

```
55,2,6,4,32,12,9,73,26,37,

2,6,4,32,12,9,55,26,37,73,
2,4,6,12,9,32,26,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
```

```
    2,4,6,9,12,26,32,37,55,73,
    2,4,6,9,12,26,32,37,55,73,
    2,4,6,9,12,26,32,37,55,73,
    2,4,6,9,12,26,32,37,55,73,
```

排序过程是用嵌套的 for 循环完成的。对 10 个元素的数组，一共进行 9 轮比较（pass 从 1～9）。每轮要进行 size－pass 次比较，以决出一个最大值。

程序首先进行第一轮比较（pass＝1），即先比较 a[0]和 a[1]，再比较 a[1]和 a[2]，然后比较 a[2]和 a[3]，……，直到比较完 a[8]和 a[9]为止。数组有 10 个元素，但只比较了 9 次（0～8）。因为是朝一个方向两个两个连续比较的，较大的值在比较之后总是放在二者的后面。9 次比较使 10 个元素都参与了比较，并在第一轮比较完后，最大的值“下沉”到数组底部，即 a[9]。

程序然后进行第二轮比较（pass＝2），因为最后一个元素 a[9]已经确定为最大，无须与前面的元素比较，比较只须在前面 9 个元素中进行，所以该轮只需 8 次比较（0～7）。比较完第二轮后，次大的数组元素“下沉”到 a[8]。

程序再进行第三轮比较（pass＝3），该轮比较了 7 次，确定了 a[7]。

直到程序进行第九轮比较（pass＝9），该轮比较只进行 1 次，以确定 a[0]和 a[1]中的大者放在 a[1]中。

从结果看出，经过 3 轮比较，就将数组从小到大排序完毕。只有在最坏情况，即全部元素从大到小排列时，才需要全部 9 轮的排序。

冒泡排序法比较易于实现，但是不论情况好坏，都要进行所有轮的比较，运行速度较慢。

2. 插入排序法（insert sort）

插入排序法是一个简单，但相对比较高效的排序算法。

插入排序通过把数组中的元素插入到适当的位置来进行排序。步骤为：

(1) 将数组中的前两个元素按排序顺序排列。

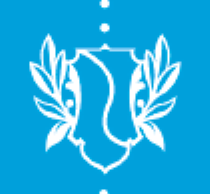
(2) 把下一个元素（第 3 个）插入到其对应于已排序元素的排序位置。

(3) 对于数组中的每个元素重复(2)。即把第 4 个元素插入到适当位置，然后是第 5 个元素，等等。

例如，对于 WVLMABCONP 字母序列的一个插入排序见图 7-6。

原数据	第一轮	第二轮	第三轮	第四轮	第五轮	第六轮	第七轮	第八轮	第九轮
W	**V**	**L**	**L**	**A**	**A**	**A**	**A**	**A**	**A**
V	**W**	**V**	**M**	**L**	**B**	**B**	**B**	**B**	**B**
L	L	**W**	**V**	**M**	**L**	**C**	**C**	**C**	**C**
M	M	M	**W**	**V**	**M**	**L**	**L**	**L**	**L**
A	A	A	A	**W**	**V**	**M**	**M**	**M**	**M**
B	B	B	B	B	**W**	**V**	**O**	**N**	**N**
C	C	C	C	C	C	**W**	**V**	**O**	**O**
O	O	O	O	O	O	O	**W**	**V**	**P**
N	N	N	N	N	N	N	N	**W**	**V**
P	P	P	P	P	P	P	P	P	**W**

图 7-6　插入排序法



图中加黑的字体部分是已经排好序的。

第一轮,是 W 和 V 比较,按升序,则交换位置。

第二轮,L 插入到已排序的 VW 中,形成 LVW。L 先跟最大的 W 比较,顺便将 W 移到 L 的位置,再与 V 比较,将 V 移到原 W 的位置,原 V 的位置放 L。

第三轮,M 插入到已排序的 LVW 中,形成 LMVW。一边比较,一边移动位置,发现 M 比 L 大时,立即停止该轮的比较,因为 M 的位置已经找到。

第四轮,类似的过程。

直到第九轮全部排序完毕。

所谓插入过程,即待插元素与左边元素不断比较及挪移的过程。若小于,则左边元素复制(挪移)到右边,若不小于,则将待插元素安顿在右边而结束。

例如,下面的程序在主函数中调用一个整数插入排序函数:

```
//----------------------
//     ch7_10.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void isort(int * a, int size);
//----------------------
int main(){
  int array[] = {55,2,6,4,32,12,9,73,26,37};
  int len = sizeof(array)/sizeof(int);   //元素个数
  for(int i = 0; i < len; i++)           //原始顺序输出
    cout << array[i]<<",";
  cout <<"\n\n";
  isort(array, len);                     //调用排序函数
}//---------------------
void isort(int a[], int size){           //插入排序
  for(int i = 1; i < size; i++){         //共执行 size-1 轮
    int ins = a[i], idx = i-1;
    for( ; idx >= 0 && ins < a[idx]; idx--)
      a[idx+1] = a[idx];                 //后挪一个位置

    a[idx+1] = ins;                      //插入

    for(int j = 0; j < size; j++)        //比较一轮后输出
      cout << a[j]<<((j == i)?" | ":","); //'|'为已排未排分界线
    cout << endl;
  }
}//---------------------
```

运行结果为:

```
55,2,6,4,32,12,9,73,26,37,

2,55,|6,4,32,12,9,73,26,37,
2,6,55,|4,32,12,9,73,26,37,
2,4,6,55,|32,12,9,73,26,37,
2,4,6,32,55,|12,9,73,26,37,
2,4,6,12,32,55,|9,73,26,37,
2,4,6,9,12,32,55,|73,26,37,
2,4,6,9,12,32,55,73,|26,37,
2,4,6,9,12,26,32,55,73,|37,
2,4,6,9,12,26,32,37,55,73,|
```



排序函数 isort()中的 inserter 是待插入元素，index 是当前准备与插入元素比较的元素下标。

该插入排序算法好在边比较边挪位，找到插入点的同时，挪位工作也完成。挪位是赋值操作，不是交换操作，所以工作量减轻很多。但另一方面，插入排序的每轮比较都是不可缺少的，无法进一步优化算法。

3. 快速排序法(quick sort)

快速排序法被认为是效率较高的排序算法。

快速排序算法是建立在把数组分为许多部分的思想上。其工作过程是首先选择一个分界值，把数组分成两部分，大于等于分界值的元素集中到数组的某一部分，小于分界值的元素集中到数组的另一部分。对于分出来的两部分，又分别重复这个过程，直到整个数组被排序完毕。

例如，设给定一个字符数组 fedacb，选定 d 作为分界值，则在第一遍划分后数组将重新安排如下：

```
初  始  f  e  d  a  c  b
第 1 遍  b  c  a  d  e  f
```

在第一遍划分为 bca 和 def 两部分后，又分别对这两部分进行划分。这个过程是递归的。采用递归算法使程序相对可读。

分界值可以用两种不同的方法去确定：一种方法是随机确定；另一种方法是算出被排序部分各元素的中间值。当分界值正好等于被排序部分各元素的中间值时，排序速度最快。然而，找到这个中间值却不是一件容易事。即使在最坏情况下，即分界值完全偏向一方，取了一个极大或极小值，快速排序法的性能仍然是比较好的。

例如，下面的程序对有 10 个元素的数组用快速排序法排序，选择数组中间项的值作为分界值，虽然这样做的效果并不总是最佳，但它是既简单又快捷的办法：

```
//----------------------
//    ch7_11.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void qsort(int[],int,int);
//----------------------
int main(){
  int array[] = {55,2,6,4,11,12,9,73,26,37};
  int len = sizeof(array)/sizeof(int);
  for(int i = 0; i < len; i++)               //原始顺序输出
    cout << array[i]<<",";
  cout <<"\n";

  qsort(array,0,len - 1);                    //调用排序函数

  for(int i = 0; i < len; i++)               //排序结果输出
    cout << array[i]<<",";
```

```
    cout << endl;
}//---------------------
void qsort(int a[], int left, int right){        //快速排序法
    int pivot = a[right],l = left,r = right,temp;
    while(l < r){
        temp = a[l], a[l] = a[r], a[r] = temp;       //交换
        while(l < r && a[r]> pivot) --r;
        while(l < r && a[l]<= pivot) ++l;
    }
    temp = a[left], a[left] = a[r], a[r] = temp;    //使得 a[r]成为分界元
    if(left < r - 1) qsort(a,left,r - 1);
    if(r + 1 < right) qsort(a,r + 1,right);
}//---------------------
```

运行结果为：

```
55,2,6,4,32,12,9,73,26,37
2,4,6,9,12,26,32,37,55,73
```

快速排序法采用 3 个参数，一个为数组，另两个为数组的上下界。因为递归调用也是这样的形式，所以简化代码的书写。3 个参数的形式并不是必需的，因为数组参数传递的是地址，所以对于两个参数的函数原型：

```
qsort(int a[],int len);
```

其中的 left 为 0，right 为 len－1，递归调用改成如下即可：

```
if(left < r) qsort(&a[left],l - left);
if(r < right) qsort(&a[r + 1],right - r);
```

语句中“&”表示取该变量的地址，进一步的介绍见 8.1 节。

根据快速排序法，我们来分析上面字符数组的第一遍划分结果：

划分之前的准备，left＝0，right＝5，分界值 pivot＝a[(left＋right)/2]＝ a[2]＝'d'，并且l＝0，r＝5。

每次交换的结果如下：

```
f e d a c b      //划分之前
b e d a c f      //第一次交换结果
b c d a e f      //第二次交换结果
b c a d e f      //第三次交换结果
```

首先，由于 a[l]＝'f'> 'd'＝pivot，所以“while(a[l]<pivot) ++l;”循环语句结束，l＝0。

又由于 a[r]＝'b'< 'd'＝pivot，所以“while(a[r]> pivot) --r;”循环语句结束，r＝5。

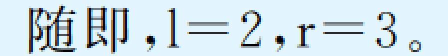
此时，a[0]与 a[5]即'f'与'b'交换，得到第一次交换结果。

随即，l＝1，r＝4。

然后，由于 a[l]＝'e'> 'd'＝pivot，所以“while(a[l]<pivot) ++l;”循环语句结束，l＝1。

由于 a[r]＝'c'< 'd'＝pivot，所以“while(a[r]> pivot) --r;”循环语句结束，r＝4。

此时，a[1]与 a[4]即'e'与'c'交换，得到第二次交换结果。



随即,l=2,r=3。

接着,由于 a[l]='d' = pivot,所以“while(a[l]<pivot) ++l;”循环语句结束,l=2。

由于 a[r]='a' < 'd'=pivot,所以“while(a[r]> pivot) --r;”循环结束,r=3。此时,a[2]与 a[3]即'd'与'a'交换,得到第三次交换结果。

随即,l=3,r=2。这时,l>r,不满足外循环条件,本次划分到此结束。

读者可以分析程序 ch7_10.cpp 中的运行结果是经过几次划分(每次划分即一次递归调用)得到的。

快速排序法在一般情况下较其他排序算法快,但是程序的理解稍微有点复杂。

7.7 数组应用:Josephus 问题

Josephus 问题是说,一群小孩围成一圈,任意假定一个数 m,从第一个小孩起,顺时针方向数,每数到第 m 个小孩时,该小孩便离开。小孩不断离开,圈子不断缩小。最后,剩下的一个小孩便是胜利者。究竟胜利者是第几个小孩呢?

解答这个问题,首先要定义一个数组,其元素个数就是小孩个数。必须预先设置一个小孩个数常量,以便定义一个数组。

对每个小孩赋以一个序号值作为小孩的标志。由于数组是局部作用域的,一旦分配之后,就删不去,要等到作用域结束才会自动抹去,所以当小孩离开时,只能修改数组元素值来标识小孩的离开。

数组是线性排列的,小孩是围成圈的,用数组表示小孩围成圈,要有一种从数组尾部跳到其头部的技巧,这就是“加 1 求模”。当数到数组尾的时候,下一个数组下标值可以算得为 0,从而回到数组首以继续整个过程。试看解答问题的程序:

```
//---------------------
// Josephus 问题解法 1
// jose1.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  const int num = 10;                    //小孩数
  int a[num];                            //建立小孩数组
  for(int i = 0; i < num; i++)           //给小孩编号
    a[i] = i + 1;
  cout <<"please input the interval: ";  //数几个小孩算一轮
  int interval;
  cin >> interval;                       //输入数一轮的间隔数
  for(int i = 0; i < num; i++)           //输出全部小孩编号
    cout << a[i]<<",";
  cout << endl;

  int i = -1;                            //数组下标(下一个值 0 就是第一个小孩的下标)
```

```
    for(int k = 1; 1; k++){                      //处理第 num - 1 轮
      for(int j = 0; j < interval; ){            //在圈中数一轮小孩
        i = (i + 1) % num;                       //对下标加 1 求模
        if(a[i]!= 0)                             //如果该元素的小孩在圈中,则承认数数有效
          j++;
      }
      if(k == num) break;
      cout << a[i]<<",";                         //输出离开的小孩之编号
      a[i] = 0;                                  //标识该小孩已离开
    }
    cout <<"\nNo."<< a[i]<<" boy've won.\n";  //输出胜利者
}//--------------------
```

运行结果为：

```
c:\> jose1
please input the interval: 8
1,2,3,4,5,6,7,8,9,10
8,6,5,7,10,3,2,9,4
No.1 boy've won.
c:\> jose1
please input the interval: 2
1,2,3,4,5,6,7,8,9,10
2,4,6,8,10,3,7,1,9
No.5 boy've won.
```

程序运行了两遍。第一遍，m 取值为 8，得到胜利者是第 1 个小孩；第二遍，m 取值为 2，得到胜利者是第 5 个小孩。

程序中，小孩数 num 用常量定义，这样数组定义的大小就可用此常量表示。用一个循环给小孩编号，依次为 1，2，3，……，不管小孩有几个，小孩的编号只与小孩个数有关。

随机输入的 m 值应大于 0，一般不能超过小孩数。读者可思考如何进行控制？

在处理离开小孩的循环前，初始化正要处理第 1 个小孩给 k，初始化数组的下标为－1，因为下一个值 0 下标表示数组第一个元素，即起始第一个小孩。

在 for 循环中的 for 循环完成数 m 个小孩的工作。数组中含有离开的和未离开的小孩，标识为 0 的是离开的小孩，否则，数组元素的值是小孩的编号。因此，往前数一下，须确认该小孩含有非 0 值。另外，下标的移动是重要的，i 值加 1 是下一个下标，但该下标有可能越过数组的边界，所以对其进行模运算就能保证下标在数组范围内循环移动。

每次处理小孩离开时，都要遍历整个数组，所以该程序效率是不高的。用数组来表示小孩围成的圈，数据结构的描述是简单的，程序语句的行数也不多，但处理是富于技巧的，理解比较费事。原始的 C 程序设计方法多与此相像。

7.8　数组应用：矩阵乘法

如果矩阵 A 乘以 B 得到 C，则必须满足如下规则：

(1) 矩阵 A 的列数与矩阵 B 的行数相等；

(2) 矩阵 A 的行数等于矩阵 C 的行数；


(3) 矩阵 B 的列数等于矩阵 C 的列数。

例如,下面的例子说明两个矩阵是如何相乘的:

$$\begin{pmatrix} 5 & 7 \\ 8 & 3 \\ 7 & 4 \end{pmatrix} \times \begin{pmatrix} 12 & 3 & 6 \\ 4 & 2 & 7 \end{pmatrix} = \begin{pmatrix} 88 & 29 & 79 \\ 108 & 30 & 69 \\ 100 & 29 & 70 \end{pmatrix}$$

在结果矩阵中,第 1 行第 1 列的元素是 88,它通过下列计算得来:

$$5 \times 12 + 7 \times 4 = 88$$

即若矩阵 $A_{mn} \times B_{nl} = C_{ml}$,则:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

其中,A_{mn}表示 $m \times n$ 矩阵,c_{ij}是矩阵 C 的第 i 行 j 列元素。

下列程序是求 $A_{34} \times B_{45} = C_{35}$ 的矩阵乘法:

```
//----------------------
//    ch7_12.cpp
//----------------------
#include <iostream>
#include <iomanip>
using namespace std;
//----------------------
int a[3][4] = {{ 5, 7, 8, 2},
               {-2, 4, 1, 1},
               { 1, 2, 3, 4}};
int b[4][5] = {{4, -2, 3, 3, 9},
               {4, 3, 8, -1, 2},
               {2, 3, 5, 2, 7},
               {1, 0, 6, 3, 4}};
int c[3][5];
//----------------------
bool mulMatrix(int a[3][4], int arow, int acol,
               int b[4][5], int brow, int bcol,
               int c[3][5], int crow, int ccol);        //c = a x b
//----------------------
int main(){
  if(mulMatrix(a,3,4, b,4,5, c,3,5)){
    cout <<"illegal matrix multiply.\n";
    return 1;
  }
  for(int i = 0; i < 3; i++){                           //输出矩阵乘法的结果
    for(int j = 0; j < 5; j++)
      cout << setw(5)<< c[i][j];
    cout << endl;
  }
}//----------------------
bool mulMatrix(int a[3][4], int arow, int acol,
               int b[4][5], int brow, int bcol,
               int c[3][5], int crow, int ccol)
{
  if(!((acol == brow)&&(crow == arow)&&(ccol == bcol)))    //正确性检查
    return 1;
```




```
    for(int i = 0; i < crow; i++)          //行
    for(int j = 0; j < ccol; j++)          //列
    for(int n = 0; n < acol; n++)          //求一个元素
      c[i][j] += a[i][n] * b[n][j];

    return 0;
  }//--------------------
```

运行结果为：

```
66   35   123   30   123
11   19    37   -5     1
22   13    58   19    50
```

该程序先定义两个全局数组，然后调用矩阵乘法函数，对应地，矩阵乘法函数的参数是3个二维数组表示的矩阵，分别有两个行列项。二维数组的大小最好也要一一对应。

矩阵乘法函数中，先对行列值进行校验，如果不符合要求，及时返回一个出错信息。在用二重循环计算矩阵结果时，又用了一个循环计算对应元素积之和。另外，由于数组c是全局的，默认值为全0，所以求结果时，语句“c[i][j]=0;”可以省略。

小结

数组是一个由若干同类型变量组成的集合，数组中特定的元素通过下标来访问。数组由连续存储单元组成，其起始地址对应于数组的第一个元素。数组名本身是地址，使得数组作为函数参数来传递从空间利用上显得合理。由于C字符串的'\0'结束符特性，所以在字符数组中，要多考虑一个字节安排该字符。

数组应用是广泛的，排序是一种典型的应用。此处的Josephus问题解是在数组中采用了一些技巧。矩阵乘法是二维数组的一个应用，内含三重循环的处理。

在C++中，数组和指针是密切相关的，下一章将讨论指针，只有读完这两章，才能透彻理解C++语言的这些构成。

练习

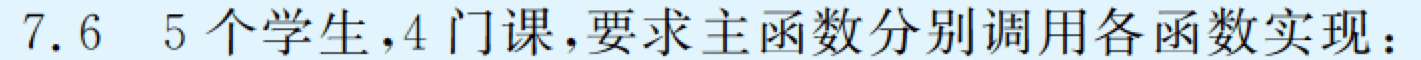
7.1 一个10个整数的数组(34,91,83,56,29,93,56,12,88,72)，找出最小数和其下标，并在主函数中打印最小数和下标。

7.2 n个数，已按从小到大顺序排列。在主函数中输入一个数，调用一个函数，它把输入的数插入原有数列中，保持大小顺序，输出插入前后的两个数组，并将被挤出的最大数(有可能就是被插入数)返回给主函数输出。

7.3 17个人围成圈，编号为1～17，从第1号开始报数，报到3的倍数的人离开，一直数下去，直到最后只剩下一人。求此人的编号。

7.4 改进ch7_9.cpp中的冒泡排序算法，使之在新一轮比较中，若没有发生元素交换，则认为已排序完毕。

7.5 输入一个n×n的矩阵，求出两条对角线元素值之和。



7.6 5个学生，4门课，要求主函数分别调用各函数实现：

(1) 找出成绩最高的学生序号和课程。

(2) 找出不及格课程的学生序号及其各门课的全部成绩。

(3) 求全部学生各门课程的平均分数，并输出。

7.7 编程求矩阵的加法：

$$\begin{pmatrix} 5 & 7 & 8 \\ 2 & -2 & 4 \\ 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 4 & -2 & 3 \\ 3 & 9 & 4 \\ 8 & -1 & 2 \end{pmatrix}$$

第8章　指针

C++语言拥有在运行时获得变量的地址和操纵地址的能力，在其他任何语言中，理解它们是如何工作的或许都不如在 C++中这么必不可少。这种用来操纵地址的特殊类型变量就是指针。指针用于数组，作为函数参数，用于内存访问和堆内存操作。指针对于成功地进行 C++语言程序设计是至关重要的。指针功能最强，但又最危险。学习本章后，要求能够使用指针，能够用指针给函数传递参数，理解指针、数组和字符串之间的紧密联系，能够声明和使用字符串数组，正确理解命令行参数，理解函数指针的用法。

8.1　指针的概念

1. 指针类型

我们学过基本数据类型，如 int、float、char、double 等，其中每一种基本数据类型都有相应的指针类型。如可以建立整型指针以处理整型数，建立字符指针以处理字符等。

2. 定义指针

在这之前，"*"是乘法符号；在这里，用作定义指针。

例如，指向整型数的指针是包含该整型数地址的变量或常量：

```
int * ip;
const int * icp;        //因为常量也具有地址,所以有指向常量的指针
```

*表示指针，int 表示该指针的类型是整型，ip 和 icp 是指针的名字。

关于指向常量的指针见 8.5 节。

又如，指向字符的指针是包含字符地址的变量或常量：

```
char * cptr;
const char * ccptr;     //指向字符常量的指针
```

char 表示该指针的类型是字符型，cptr 和 ccptr 是指针的名字。



指针的定义语句,由数据类型后跟星号,再跟随指针名组成。

指针是一个内存实体,具有值。要使用指针,就必须定义指针。指针有指针常量和指针变量之分,定义指针通常定义的是指针变量,即可以随时改变指针的指向。所以,指针与指针变量经常划等号。

通常,每个指针都有一个类型(void * 指针除外,见 8.6 节)。指针是变量,因此它与其他基本数据类型一样,凡可声明变量的地方,就可声明指针,它可以是全局、静态全局、静态局部和局部的。

上面的 ip 和 cptr 两个指针定义都分配了空间,但是都没有指向任何内容。正如定义整型或浮点变量没有给它赋值一样,指针也没有被赋值。

定义名为 iPtr 的指针可以写成:

```
int *iPtr;       //* 靠左
int *iPtr;       //* 靠右
int *iPtr;       //* 两边都不靠
```

它们表示同一个意思。**在指针定义中,一个 * 只能表示一个指针。**定义语句:

```
int *iPtr1,iPtr2;
```

表示定义一个 iPtr1 指针变量和一个 iPtr2 整型变量。如果要定义两个指针变量,须:

```
int *iPtr, *iPtr;
```

3. 建立指针

建立指针包括定义指针和给指针赋初值。

变量存在于内存中的某位置(地址)。例如,一旦有了变量,则放置该变量的地方就用内存地址描述。

用 & 操作符可以获取变量的地址,指针用于存放地址。

例如,定义整型指针 iPtr,定义整型变量 iCount,把变量 iCount 的地址赋给指针 iPtr:

```
int * iPtr;
int iCount = 18;
iPtr = &iCount;        //将地址赋给存放地址的指针
```

指针的工作方式见图 8-1。存放变量 iCount 的地址是 0000:F822,用 &iCount 表示,上例将该地址赋给指针变量 iPtr。等号右边是一个地址,其左边是一个地址左值,即 iPtr,其类型也是一个地址(整型数地址),等号两边匹配。

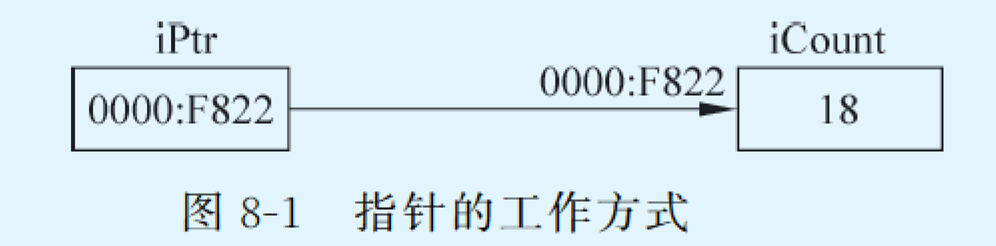


图 8-1 指针的工作方式

这时候的指针 iPtr 赋了值 0000:F822。

变量存储地为内存,其驻留位置即内存地址。内存地址在 32 位(CPU)机器中是一个 32 位二进制的整型数,书中 0000:F822 的两段式十六进制数表示法,是当时旧式 16 位机器



配置 32 位数据传输线的一种分段地址表示法。歪打正着的是,指针值总是带有类型的烙印,忌讳与纯整数相关联,这样的表示恰好可以将地址与整数表示形式相区别。

4. 间接引用指针

"*"是乘法,又可以用于定义指针,在这里可用于指针的间接引用(* 的第三个用途)。

间接引用指针时,可获得由该指针指向的变量内容。

例如,下面的程序间接引用指针 iPtr,输出 iCount 的内容:

```
//---------------------
//    ch8_1.cpp
//---------------------
#include<iostream>
using namespace std;
//---------------------
int main(){
  int *iPtr;
  int iCount = 18;
  iPtr = &iCount;
  cout << * iPtr << endl;   //间接引用指针
}//--------------------
```

运行结果为:

```
18
```

该运行结果就是指针 iPtr 所指向的变量 iCount 中的内容。

*** 放在可执行语句中的指针之前,为间接引用操作符; *放在指针定义中时,为指针定义符。**

非指针是不能用间接引用操作符的,因为 * 只能作用于地址。例如:

```
cout << *iCount;    //error: 非指针不能用间接引用操作符
```

间接引用的指针既可用于右值,也可用于左值。

例如,上例 ch8_1.cpp 中,通过 iPtr 改变变量 iCount 的原始值:

```
*iPtr = 58;           //指针的间接引用作左值
cout << iCount << endl;//将显示 58
```

改变 * iPtr 就是改变 iCount。所以,输出结果为 58。

设想图书馆的卡片,每个卡片都是一个指向书的指针,卡片中有书的位置。看到书的位置,到书库中找到该书,此时就间接地引用了那张卡片。

还书时,把书放回书架,管理员查到书架上的号码,号码与所使用的卡片号码相对应。管理员把书放到书架的固定位置,也间接引用了指针。

5. 指针的地址

指针也是变量,**是变量就具有内存地址**。所以指针也有地址。

例如,下面的程序输出 iCount 变量值,以及 iPtr 和 iCount 的地址值:

```
//---------------------
//    ch8_2.cpp
//---------------------
```

```
#include <iostream>
using namespace std;
//---------------------
int main(){
  int iCount = 18;
  int *iPtr = &iCount;
  *iPtr = 58;

  cout << iCount << endl;
  cout << iPtr << endl;
  cout << &iCount << endl;      //与 iPtr 值相同
  cout << *iPtr << endl;        //与 iCount 值相同
  cout << &iPtr << endl;        //指针本身的地址
}//---------------------
```

运行结果为：

```
58
0x0067fe00
0x0067fe00
58
0x0067fdfc
```

0x0067fe00 是指针 iPtr 的值，即变量 iCount 的地址。0x0067fdfc 是指针 iPtr 存放的地址。二者是有区别的，见图 8-2。注意地址 0067:FDFC 和 0x0067fdfc 仅大小写不同，是同一个地址的两种不同表示。

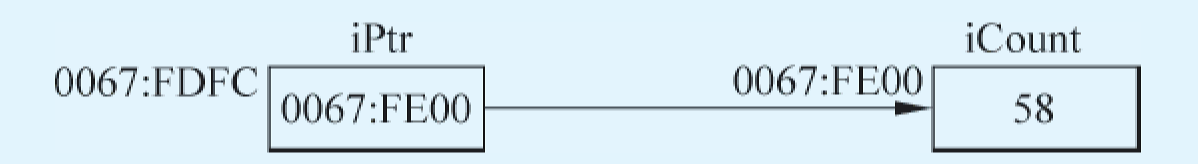


图 8-2　指针值与指针的地址值是不同的

***iPtr 的类型是整型，指针 iPtr 指向该整数，iPtr 的类型是整型指针，而 iPtr 的地址(即 &iPtr)的类型是整型指针的地址，即指向整型指针的指针。三者都不相同。**在 8.8 节中将会看到，**指针的地址就是二级指针**。

有了取地址操作符 & 和间接引用操作符 * 后，我们有 6 种对变量的操作，并且应该理解这 6 种操作的意义。它们是 iPtr、iCount、*iCount、*iPtr、&iPtr 和 &iCount。其中，*iCount 是非法的。如果访问 *iCount，BC 将报告"无效的间接引用"(invalid indirection)错误，VC 将报告"非法间接引用"(illegal indirection)错误。

6. 指针与整型数的区别

指针在使用中必须类型匹配。例如：

```
int iCount = 26;
int *iPtr = &iCount;          //定义语句：*在此处作定义指针用，而非间接引用
*iPtr = &iCount;              //error:不能将整型地址转换成整型数
*iPtr = 50;                   //执行语句：*在此处作间接引用
```

例中前面两句是定义语句，后面两句是执行语句。*iPtr 的类型是整型，&iCount 的类型是整型指针，**指针值不是整型数**，所以赋值语句"*iPtr=&iCount;"在 BC 中会引起类型



转换的错误(cannot convert int * to int)。

在 32 位机器中,整数和指针都占 4 个字节,内存表示的方式也都是二进制整数,但指针和整数表示的是不同的类型。

可以强制转换。例如,允许语句“* iPtr=(int)&iCount;”,但要注意其赋值的意义。该语句表示将变量 iCount 的地址值作为一个整型数赋给 * iPtr,即 iCount 变量。

7. 指针的初始化

普通变量在定义时可以初始化,指针也可在定义时初始化。**指针初始化的值是该指针类型的地址值**。例如:

```
int iCount = 26;
int * iPtr = &iCount;     //初始化为整型地址
*iPtr = &iCount;          //error
```

不要将“int * iPtr= &iCount;”与“* iPtr= &iCount;”混淆。前者是定义语句,* 是指针定义符,C++为 iPtr 指针分配一个指针空间,并用 iCount 的地址值初始化;后者是赋值语句,左右两边类型不匹配。

* 操作符在指针上的两种用途要区分开:定义或声明时,建立一指针;执行时,间接引用一指针。

指针在使用前,要进行初始化。例如,下面的代码是危险的:

```
int count;
int * iPtr;
*iPtr = 58;    //!
```

指针忘了赋值比整型变量忘了赋值危险得多。

iPtr 当前指向什么地方?该代码能通过编译,但没有赋初值的指针 iPtr 是一个随机地址。“* iPtr=58;”是把 58 赋到内存中的随机位置,因此将改写另一存储位置的数值,甚至修改了栈中的函数返回地址,计算机将死机或进入死循环。

8. 指针类型与实际存储的匹配

指针是有类型的,给指针赋值,不但必须是一个地址,而且应该是一个与该指针类型相符的变量或常量的地址。

例如,下面的代码错将浮点类型的变量地址赋给整型指针:

```
//---------------------
//    ch8_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  float f = 34.5;
  float * fPtr = &f;       //浮点指针
  int * iPtr = (int *)&f;  //warning:将浮点变量的地址赋给整型指针

  cout << f << endl
```

```
        <<"iPtr:"<< iPtr <<" = >"<< *iPtr <<"\n"
        <<"fPtr:"<< fPtr <<" = >"<< * fPtr <<"\n\n";
  *iPtr = * fPtr;      //隐式数据转换
  cout << f << endl
        << *iPtr << endl
        << * fPtr << endl;
}//--------------------
```

运行结果为：

```
34.5
iPtr:0x0067:fdfc = > 1107951616
fPtr:0x0067:fdfc = > 34.5

4.76441e-44
34
4.76441e-44
```

该程序在BCB6(32位机器环境)中运行。程序中，定义整型指针iPtr时，赋给一个指向浮点变量f的地址。在BCB中，“int * iPtr=&f;”和“iPtr=fPtr;”都会引起“可疑的指针转换”(suspiciuos pointer conversion)警告。在VC中该程序通不过编译，将给出“不能将浮点指针转换成整型指针”(cannot convert from 'float * ' to 'int * ')的错误。

输出f的内容与输出 * fPtr的内容是一致的，但是输出 * iPtr的内容与 * fPtr不一致，尽管其地址是相同的，但所表示的类型不同。**iPtr是整型指针，它总是访问该地址中的整型数；而fPtr是浮点指针，总是访问该地址的浮点数**。

“ * iPtr= * fPtr;”是将浮点数赋给整型变量，它是合法的语句，但会引起隐式类型转换(见3.3节)，截断小数位数，失去一定的精度。在赋值中，C++自动进行数据类型转换，得到整型数34。由于类型不一致，在VC中将给出“转换浮点到整型”(conversion from 'float' to 'int')的警告。

fPtr与iPtr所表示的地址值在不同的机器运行环境中是不同的，本处是iPtr::0x0067:fdf7。由于fPtr与iPtr都指向同一地址，所以赋值之后，在该地址上，浮点数被整型数34覆盖了。此时，若要按浮点方式来读取整型数位模式下的34，则会得到一个“意想不到”(若知道浮点数的表示方法，则可推算出该浮点数的值)的浮点数。

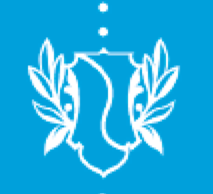
程序运行的结果证实，iPtr并不单纯指向f的地址。* **iPtr中的内容是起始点为f的地址的一个整型数**。

指针具有一定类型，它是值为地址的变量，该地址是内存中另一个该类型变量的存储位置。或者说指针是具有某个类型的地址。

8.2 指针运算

指针可以进行加减运算。加减运算的结果，指针挪移到了邻近的内存单元，于是指针的运算与数组扯上了关系。

数组名本身，没有方括号和下标，它实际上是地址，表示数组起始地址。整型数组的数组名本身得到一整数地址，字符数组的数组名得到一字符型地址。



可以把数组起始地址赋给一指针,通过移动指针(加减指针)来对数组元素进行操作。例如,下面的程序用指针运算来计算数组元素的和:

```
//---------------------
//    ch8_4.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  int iArray[10];
  int sum = 0;
  int * iPtr = iArray;              //用数组名 iArray 给指针初始化

  for(int i = 0; i < 10; i++)       //给数组赋值
    iArray[i] = i * 2;
  for(int idx = 0; idx < 10; idx++){  //累计数组元素
    sum += *iPtr;
    iPtr++;
  }
  cout << "sum is " << sum << endl;
}//---------------------
```

运行结果为:

```
sum is 90
```

指针 iPtr 被初始化为数组起始地址,因此,上例的指针定义语句“int * iPtr=iArray;”中,左右两边的类型是匹配的。它可以写成:

```
int * iPtr;
iPtr = iArray;
```

其中,“iPtr=iArray;”还可以改写成:

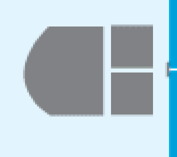
```
iPtr = &iArray[0];    //同样表示数组的第一个元素的地址
```

在程序例中,循环重复 10 次,指针遍历数组每个元素。假设在 16 位机器上,数组起始地址是 0x00000100,则指针 iPtr 的值似乎是(实际不是这样):

```
0x00000100
0x00000101
0x00000102
0x00000103
0x00000104
...
```

但我们知道 16 位机器中,整数是占两个字节的,所以数组元素的地址应该是:

```
0x00000100
0x00000102
0x00000104
0x00000106
0x00000108
...
```



由于指针是具有某个数据类型的地址，所以指针运算都是以数据类型为单位展开的。即，iPtr 是个整型指针，iPtr++使指针指向下一个整型数。因而，iPtr 的地址值其实增加了 2，见图 8-3。

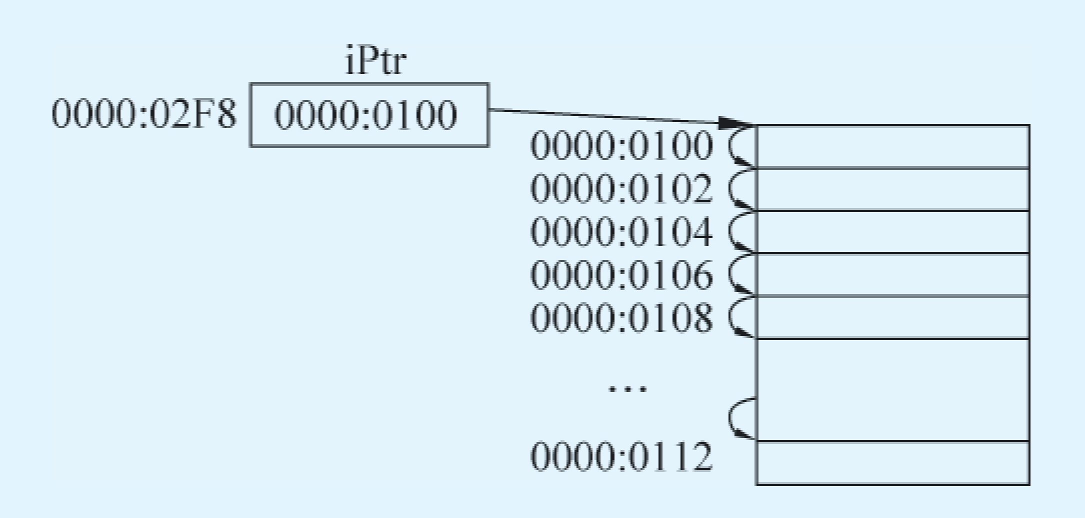


图 8-3　指针 iPtr++的移动

例如，下面的程序显示了指针移动时其地址的变化和指向的内容：

```
//---------------------
//    ch8_5.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  int iArray[10];
  int * iPtr = iArray;                  //用数组名 iArray 给指针初始化

  for(int i = 0; i < 10; i++)           //给数组赋值
    iArray[i] = i * 2;
  for(int idx = 0; idx < 10; idx++,iPtr++)  //累计数组元素
    cout <<"&Array["<< idx <<"]:"<< iPtr
         <<" =>"<< *iPtr << endl;
}//---------------------
```

运行结果为：

```
&iArray[0]:0x0067fdd4 => 0
&iArray[1]:0x0067fdd8 => 2
&iArray[2]:0x0067fddc => 4
&iArray[3]:0x0067fde0 => 6
&iArray[4]:0x0067fde4 => 8
&iArray[5]:0x0067fde8 => 10
&iArray[6]:0x0067fdec => 12
&iArray[7]:0x0067fdf0 => 14
&iArray[8]:0x0067fdf4 => 16
&iArray[9]:0x0067fdf8 => 18
```

只有加法和减法可用于指针运算。

在 16 位机器上，浮点数占 4 字节，长整数占 4 字节，字符占 1 字节，双精度数占 8 字节，所以：

对浮点型指针加 6 实际加 24；

对长整型指针加 5 实际加 20；



对字符型指针加 7 实际加 7;

对双精度型指针加 2 实际加 16。

在 ch8_4.cpp 中的循环内,语句:

```
sum += *iPtr;
iPtr ++ ;
```

可压缩成一条语句:

```
sum += *(iPtr ++ );
```

以更有效地处理数组。由于++与 * 的操作符优先级相同,它们是右结合的,所以括号可以省略。即可表示为:

```
sum += *iPtr ++ ;
```

"*iPtr++"这种形式的表达式在 C++中很常见。

指针减法的原理与加法相同。只是,不论指针的加法还是减法,其访问操作都必须是有意义的,否则是危险的。例如:

```
int a[5];
int *iPtr = &a[1];
iPtr -- ;            //指向 &a[0]
*iPtr = 3;           //ok
iPtr -- ;            //指向 &a[ - 1],dangerous!
*iPtr = 6;           //damage!
```

8.3 指针与数组

数组名可以拿来初始化指针,数组名就是数组第一个元素地址。即对于数组 a,有 a 等于 &a[0]。此外,对于:

```
int a[100];
int * iPtr = a;
```

我们有第 i 个元素:

```
a[i]  等价于  *(a + i)  等价于  iPtr[i]  等价于  *(iPtr + i)
```

a[i]表示数组的第 i 个元素的值,而 a+i 表示第 i 个元素的地址,对其间接访问,即 *(a+i)就表示第 i 个元素的值。另外,**下标操作是针对地址而不仅仅是针对数组名的**,所以 iPtr[i]也表示第 i 个元素的值。上面 4 个式子等价的事实使得数组与指针相互转换非常灵活。

不但如此,相应地,我们还有第 i 个元素的地址:

```
&a[i]  等价于  a + i  等价于  iPtr + i  等价于  &iPtr[i]
```

数组名本身是一指针,它的类型是指向数组元素的指针。&a[i]表示数组第 i 个元素的地址。这 4 式的等价与前面 4 式的等价是对应的。

例如,对于前面数组的求和运算,可以有下面 5 种方法:

```
//----------------------
//    ch8_6.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int iArray[] = {1,4,2,7,13,32,21,48,16,30};      //全局数组
//----------------------
int main(){
  int sum1 = 0,sum2 = 0,sum3 = 0,sum4 = 0,sum5 = 0; //存放每种方法的结果
  int size = sizeof(iArray)/sizeof( *iArray);    //元素个数
  for(int n = 0; n < size; n++)                  //方法 1
    sum1 += iArray[n];

  int * iPtr = iArray;
  for(int n = 0; n < size; n++)                  //方法 2
    sum2 += *iPtr++;

  iPtr = iArray;                                 //此句不能省略,因为方法 2 修改了 iPtr
  for(int n = 0; n < size; n++)                  //方法 3
    sum3 += *(iPtr + n);

  iPtr = iArray;                                 //此句可以省略,因为方法 3 没有修改 iPtr
  for(int n = 0; n < size; n++)                  //方法 4
    sum4 += iPtr[n];

  for(int n = 0; n < size; n++)                  //方法 5
    sum5 += *(iArray + n);

  cout << sum1 << endl
      << sum2 << endl
      << sum3 << endl
      << sum4 << endl
      << sum5 << endl;
}//--------------------
```

运行结果为：

```
174
174
174
174
174
```

程序中求元素个数的表达式中，sizeof(*iArray)表示数组元素类型所占的字节数，它可以表示为 sizeof(int)。没有这样做的原因是，该语句可以适应 float 或 double 等类型的数组，任凭全局数据的类型如何变化，主函数中的语句都不用作修改。

一般来说，在机器指令的实现上，指针表示不比下标表示效率更低。所以，*(iPtr+n)(指针表示)或 *(iArray+n)的表示总是不差于 iPtr[n](下标表示)或 iArray[n]的表示，但从可读性来说，后者似乎优于前者。

数组名是指针常量，区别于指针变量，所以，给数组名赋值是错误的。

例如，下面的代码对数组求和，企图以数组名的增量来实现：

```
int iArray[100];
int sum = 0;
//…
for(int i = 0; i < 100; i ++ )
{
   sum += iArray;
   iArray ++ ;      //error:数组名不是左值
}
```

由于**指针常量不是左值**,“iArray++;”意味着“iArray=iArray+1;”,即要求 iArray 是一个左值。所以,上例中,BC 编译器将给出“Lvalue required”的错误,VC 编译器将给出“left operand must be an lvalue”的错误。

对于编译器来说,**数组名表示内存中分配了数组的固定位置,修改了这个数组名,就会丢失数组空间**,所以数组名所代表的地址不能被修改。

8.4 堆内存分配

1. 堆内存

堆(heap)是内存空间。堆是区别于栈区、全局数据区和代码区的另一个内存区域。**堆允许程序在运行时(而不是在编译时)申请某个大小的内存空间。**

在通常情况下,一旦定义了一个数组,那么不管这个数组是局部的(在栈中分配)还是全局的(在全局数据区分配),它的大小在程序编译时即是已知的,因为必须用一个常数对数组的大小进行声明:

```
int i = 10;
//...
int a[i];     //error: 定义时不允许数组元素个数为变量
int b[20];    //ok
```

但是,在编写程序时不是总能知道数组应该定义成多大,如果数组元素定义多了就会浪费内存,更何况有时候根本不知道需要使用多少个数组元素。因此,需要在程序运行时从系统中获取内存。

程序在编译和连接时不予确定这种在运行中获取的内存空间,这种内存需求随着程序运行的进展而时大时小,这种运行中申请的内存就是堆内存,所以堆内存是动态的。堆内存也称动态内存。

2. 获得堆内存

函数 malloc()是 C 程序获得堆内存的一种方法,它在 stdlib.h 或 cstdlib 头文件中声明。malloc()函数的原型为:

```
void * malloc(size_t size);
```

size_t 即 unsigned long。

该函数从堆内存中“切下”一块 size 大小的内存,将指向该内存的地址返回。该内存中的内容是未知的。

例如，下面的程序从堆中获取一个整数数组，赋值并打印：

```
//----------------------
//    ch8_7.cpp
//----------------------
#include <iostream>
#include <cstdlib>                              //用到 malloc()
using namespace std;
//----------------------
int main(){
  cout <<"please input a number of array:\n";
  int aSize;                                    //元素个数
  cin >> aSize;

  int * ap = (int * )malloc(aSize * sizeof(int));   //堆内存分配

  for(int cnt = 0; cnt < aSize; cnt++)
    ap[cnt] = cnt * 2;
  for(int cnt = 0; cnt < aSize; cnt++)
    cout << ap[cnt]<<" ";

  cout << endl;
}//--------------------
```

运行结果为：

```
C>ch8_7
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

程序编译和连接时，在栈中分配了 aSize 整型变量和 ap 整型指针空间。程序运行中，调用函数 malloc()并以键盘输入的整数值作为参数。malloc()函数在堆中寻找未被使用的内存，找够所需的字节数后返回该内存的起始地址。因为 malloc()函数并不知道用这些内存干什么，所以它返回一个没有类型的指针(见 8.6 节)。但对整数指针 ap 来说，malloc()函数的返回值必须显式转换成整数类型指针才能被接受(ANSI C++标准)。

一个拥有内存的指针完全可以被看作一个数组，而且位于堆中的数组和位于栈中的数组结构是一样的。在 ch8_7.cpp 的代码中，ap 申请到堆内存后，在循环赋值中的表达式“ap[cnt]=cnt * 2;”正是这样应用的。

上例中并没有保证一定可以从堆中获得所需内存。有时，系统能提供的堆空间不够分配，这时系统会返回一个空指针值 NULL。这时所有对该指针的访问都是破坏性的，因此调用 malloc()函数更完善的代码应该如下：

```
if(ap = (int * )malloc(aSize * sizeof(int)) == NULL)
{
   cout <<"can't allocate more memory, terminating.\n";
   exit(1);
}
```

3. 释放堆内存

我们把堆看作可以按要求进行分配的资源或内存池。程序对内存的需求量随时会增大

或缩小。程序在运行中可能经常会不再需要由 malloc()函数分配的内存,而且程序还未运行结束,这时就需要把先前所占用的内存释放回堆以供程序的其他部分使用。

函数 free()返还由 malloc()函数分配的堆内存,其函数原型为:

```
void free(void*);
```

free()参数是先前调用 malloc()函数时返回的地址。把其他值传给 free()很可能会造成灾难性的后果。

例如,下面的程序完善程序 ch8_7.cpp:

```
//--------------------
//    ch8_8.cpp
//--------------------
#include <iostream>
#include <cstdlib>        //用到 malloc()
using namespace std;
//--------------------
int main(){
  cout<<"please input a number of array:\n";
  int aSize;              //元素个数
  cin>>aSize;
  int* ap = (int*)malloc(aSize*sizeof(int));
  if(ap){
    cout<<"can't allocate more memory, terminating.\n";
    return 1;
  }
  for(int cnt = 0; cnt<aSize; cnt++)
    ap[cnt] = cnt*2;
  for(int cnt = 0; cnt<aSize; cnt++)
    cout<<ap[cnt]<<" ";
  cout<<endl;
  free(ap);               //释放堆内存
}//--------------------
```

运行结果为:

```
C>ch8_8
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

4. new 与 delete

new 和 delete 是 C++专有的操作符,它们不用头文件声明。new 类似于函数 malloc(),分配堆内存,但比 malloc()更简练。new 的操作数为数据类型,它可以带初始化值表或单元个数。new 返回一个具有操作数的数据类型的指针。

返还 delete 类似于函数 free(),释放堆内存。delete 的操作数是 new 返回的指针,当返还的是 new 分配的数组时,应该带[]。

例如,下面的程序是程序 ch8_8.cpp 的 C++堆内存申请版:

```
//----------------------
//    ch8_9.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  cout <<"please input a number of array:\n";
  int aSize;                    //元素个数
  cin >> aSize;
  int * array = new int[aSize];    //分配堆内存

  for(int i = 0; i < aSize; i++)
    array[i] = i * 2;
  for(int i = 0; i < aSize; i++)
    cout << array[i]<<" ";

  cout << endl;
  delete[] array;              //释放堆内存
}//----------------------
```

运行结果为：

```
C > ch8_9
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

从此例可以看到，new 的返回值无须显式转换类型，直接赋给整数指针 array。new 的操作数是 int[aSize]，它只要指明什么类型和要几个元素就可以了，它比函数 malloc()更简捷。虽然 new 和 delete 在性能上略逊于函数 malloc()和 free()，但却更安全并具有更丰富的功能。在第 14 章将进一步展开 new 和 delete 的讨论。

8.5 const 指针

对于下面涉及指针定义和操作的语句：

```
int a = 1;
int * pi;
pi = &a;
* pi = 58;
```

可以看到，一个指针涉及两个变量：指针本身 pi 和指向的变量 a。修改这两个变量的对应操作为“pi=&a;”和“*pi=58;”。

1. 指向常量的指针（常量指针）

在指针定义语句的类型前加 const，表示指向的对象是常量。例如：

```
const int a = 78;
const int b = 28;
```

```
int c = 18;
const int * pi = &a;        //指针类型前加 const
*pi = 58;                   //error: 不能修改指针指向的常量
pi = &b;                    //ok:指针值可以修改
*pi = 68;                   //error: 同上
pi = &c;                    //ok:指针可以指向变量
*pi = 88;                   //error: 同上
c = 98;                     //ok
```

a 是常量,将 a 的地址赋给指向常量的指针 pi,使 a 的常量性有了保证。如果企图修改 a,则会引起"不能修改常量对象"(Cannot modify a const object)的编译错误。

可以将另一个常量地址赋给指针"pi=&b;"(指针值可以修改),这时,仍不能进行"*pi=68;"的赋值操作,从而保护了被指向的常量不被修改,见图 8-4,图中阴影部分表示不能被修改。可以将一个变量地址赋给指针"pi=&c;",这时,由于不能进行"*pi=88;"的赋值操作,从而保护了被指向的变量在指针操作中不被修改。**定义指向常量的指针只限定指针的间接访问只能读而不能写,而没有限定指针值的读写访问性。**

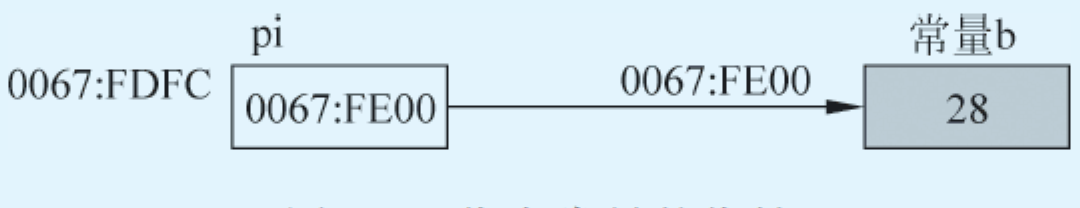


图 8-4　指向常量的指针

又如,变量 c 可以修改,这在函数传递中经常被使用。

下面的程序将两个一样大小的数组传递给一个函数,让其完成复制字符串的工作,为了防止作为源数据的数组遭到破坏,声明该形参为常量字符串:

```
//---------------------
//    ch8_10.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
void mystrcpy(char * dest, const char * source){
  while(*dest++ = *source++);
}//--------------------
int main(){
  char a[20] = "How are you!";
  char b[20];
  mystrcpy(b,a);
  cout << b << endl;
}//--------------------
```

运行结果为:

```
How are you!
```

变量字符串 a 传递给函数 mystrcpy()中的 source,使之成为常量,不允许进行任何修改。但在主函数中,a 却是一个普通数组,没有任何约束,可以被修改。

由于数组 a 不能直接赋值给 b(即不允许 b=a),所以通过一个函数实现将数组 a 的内容复制给数组 b。



函数 mystrcpy()中的语句是一个空循环,条件表达式是一个赋值语句。随着一个赋值动作,便将一个数组 a 中的字符赋给了数组 b 中对应的元素,同时两个数组参数都进行增量操作以指向下一个元素。只要所赋的字符值不是'\0'(条件表达式取假值),则循环就一直进行下去。

常量指针定义"const int * pi=&a;"告诉编译,* pi 是常量,不能将 * pi 作为左值进行操作。

2. 指针常量

在指针定义语句的指针名前加 const,表示指针本身是常量。例如:

```
char * const pc = "asdf";     //指针名前加 const 定义指针常量
pc = "dfgh";                  //error: 指针常量不能改变其指针值
*pc = 'b';                    //ok: pc 内容为"bsdf"
*(pc + 1) = 'c';              //ok: pc 内容为"bcdf"
*pc ++= 'y';                  //error: 指针常量不能改变其指针值
const int b = 28;
int * const pi = &b;          //error: 不能将 const int * 转换成 int *
```

pc 是指针常量,**在定义指针常量时必须初始化**,就像常量初始化一样。这里初始化的值是字符串常量的地址,见图 8-5。



图 8-5 指向变量的指针常量图

由于 pc 是指针常量,所以不能修改该指针值。"pc="dfgh";"将引起一个"不能修改常量对象"(Cannot modify a const object)的编译错误。

pc 所指向的地址中存放的值并不受指针常量的约束,即 * pc 不是常量,所以" * pc='b';"和" * (pc+1)='c';"的赋值操作是允许的。但" * pc++ ='y';"是不允许的,因为该语句修改 * pc 的同时也修改了指针值。

由于此处 * pc 是不受约束的,所以,将一个常量的地址赋给该指针"int * const pi=&b;"是非法的,它将导致一个不能将 const int * 转换成 int * 的编译错误,因为那样将使修改常量(如" * pc=38;")合法化。

指针常量定义"int * const pc=&b;"告诉编译,pc 是常量,不能作为左值进行操作,但是允许修改间接访问值,即 * pc 可以修改。

3. 指向常量的指针常量(常量指针常量)

可以定义一个指向常量的指针常量,它必须在定义时进行初始化。例如:

```
const int ci = 7;
int ai;
const int * const cpc = &ci;    //指向常量的指针常量
const int * const cpi = &ai;    //ok
cpi = &ci;                      //error: 指针值不能修改
*cpi = 39;                      //error: 不能修改所指向的对象
ai = 39;                        //ok
```

cpc 和 cpi 都是指向常量的指针常量,它们既不允许修改指针值,也不允许修改 * cpc 的值,见图 8-6。



图 8-6　指向常量的指针常量

如果初始化的值是变量地址(如 &ai),那么不能通过该指针来修改该变量的值。也即"* cpi=39;"是错误的,将引起"不能修改常量对象"(Cannot modify a const object)的编译错误。但"ai=39;"是合法的。

常量指针常量定义"const int * const cpc= &b;"告诉编译,cpc 和 * cpc 都是常量,它们都不能作为左值进行操作。

考虑到安全性,指针一旦初始化或者赋了初值后,不轻易改动,于是指针访问数组中的元素便多用下标而不是频繁修改指针的指向。绝大多数指针应用是指针常量和常量指针常量。这便是 C++引用(第 9 章介绍)设计的缘起。

8.6　指针与函数

1. 传递数组的指针性质

一旦把数组作为参数传递到函数中,则在栈上定义了指针,可以对该指针进行递增、递减操作。

例如,下面的代码传递一个数组,并对其进行求和运算:

```
//---------------------
//    ch8_11.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
void Sum(int array[], int n){
  int sum = 0;
  for(int i = 0; i < n; i++){
    sum += * array;
    array++;      //ok:array 是一个指针,可以作为左值
  }
  cout << sum << endl;
}//--------------------
int main(){
  int a[10] = {1,2,3,4,5,6,7,8,9,10};
  Sum(a,10);
}//--------------------
```

运行结果为:

```
55
```

在主函数中,对 Sum()函数进行了调用。传递的数组参数在 Sum()中,实质上是一个指



针，所以**声明 Sum(int array[],int n)与 Sum(int * array,int n)是等价的**。调用的示意见图 8-7。

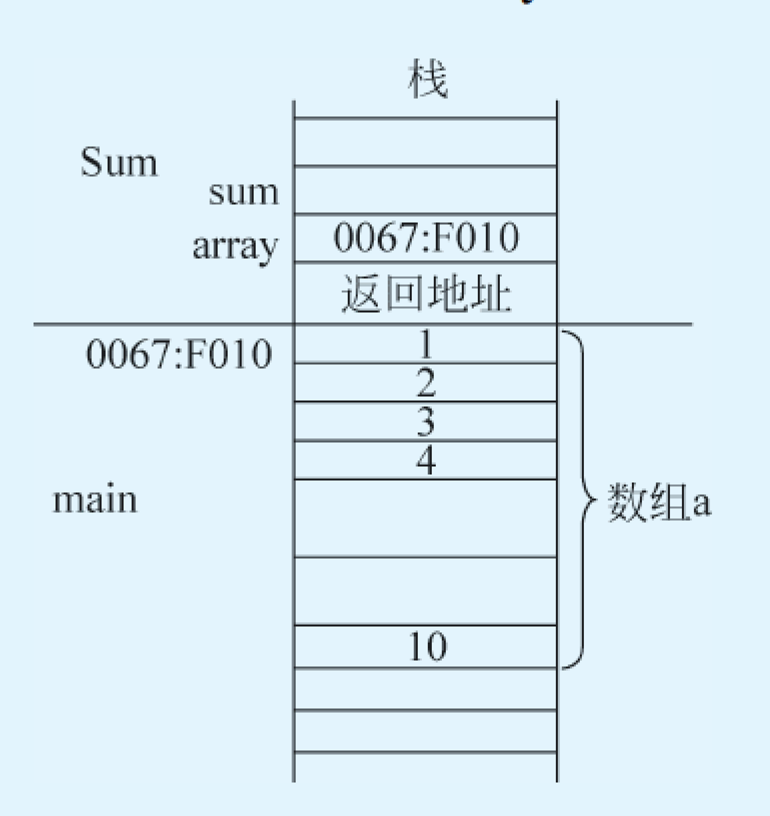


图 8-7　数组作为参数的函数调用

由于形参 array 是指针而不是数组，所以它所占的空间是指针大小而不是数组空间大小。不能用 sizeof(array)/sizeof(* array)来求取数组元素个数，这就是第二参数 n(表示数组元素个数)必须要给的原因。

尽管形参中说明的形式是 array[]数组的形式，但 C++已经明确告诉作为数组传递的参数就是指针，所以 array 可以作为左值进行 array++运算。

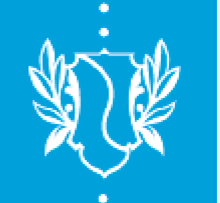
2. 使用指针修改函数参数

通过对函数的调用，函数返回一个值。然而，在很多情形下，希望函数返回不止一个值。例如，下面是想通过 swap()函数调用来交换两个整数值的程序：

```
//----------------------
//     ch8_12.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void swap(int,int);
//----------------------
int main(){
  int a = 3, b = 8;
  cout <<"a = "<< a <<", b = "<< b << endl;
  swap(a,b);
  cout <<"after swapping...\n";
  cout <<"a = "<< a <<", b = "<< b << endl;
}//---------------------
void swap(int x,int y){      //交换两个形参
  int temp = x;
  x = y;
  y = temp;
}//---------------------
```

运行结果为：

```
a = 3, b = 8
after swapping...
a = 3, b = 8
```



该 swap()函数无法返回更多的值,而且也无法改变 a 和 b 的值。由于函数参数值的传递是实参到形参的复制,被调函数内部对形参的修改并不反映到调用函数的实参中,致使 swap()中 x 和 y 作了交换,而 main()中 a 和 b 并没有交换。该函数的内存栈结构见图 8-8。

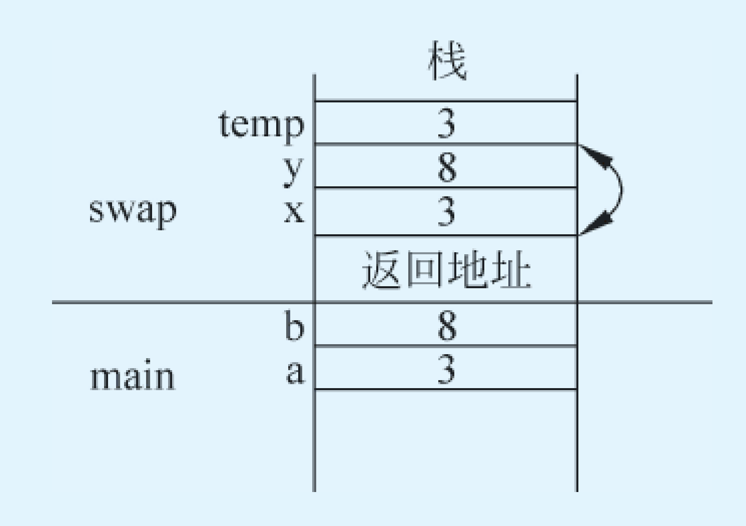


图 8-8　传递整数值的内存布局

传递指针可以使函数"返回"更多的值。这里的"返回"不是函数返回类型描述返回值的返回,而是反映了调用函数中的变量给被调函数修改了。

例如,下面的程序通过传递指针来实现整数值的交换:

```
//----------------------
//    ch8_13.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void swap(int*, int*);
//----------------------
int main(){
  int a=3, b=8;
  cout<<"a="<<a<<", b="<<b<<endl;
  swap(&a,&b);
  cout<<"after swapping...\n";
  cout<<"a="<<a<<", b="<<b<<endl;
}//---------------------
void swap(int* x,int* y){
  int temp=*x;
  *x=*y;
  *y=temp;
}//---------------------
```

运行结果为:

```
a=3, b=8
after swapping...
a=8, b=3
```

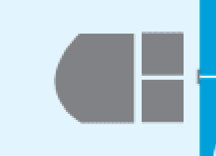
传递指针的函数调用实现过程为:

(1) 函数声明中指明指针参数,即本例中的 void swap(int * x,int * y)。

(2) 函数调用中传递变量的地址,即本例中的 swap(&a,&b)。

(3) 函数定义中对形参进行间接访问。

对 * x 和 * y 的操作,实际上就是访问调用函数的变量 a 和 b。通过函数中的局部变量



temp 的中介，使变量 a 和 b 的值被修改。调用时，给予 a 和 b 的地址作为参数，swap()函数运行后，“返回”a 和 b 的值。该函数的内存栈结构见图 8-9。

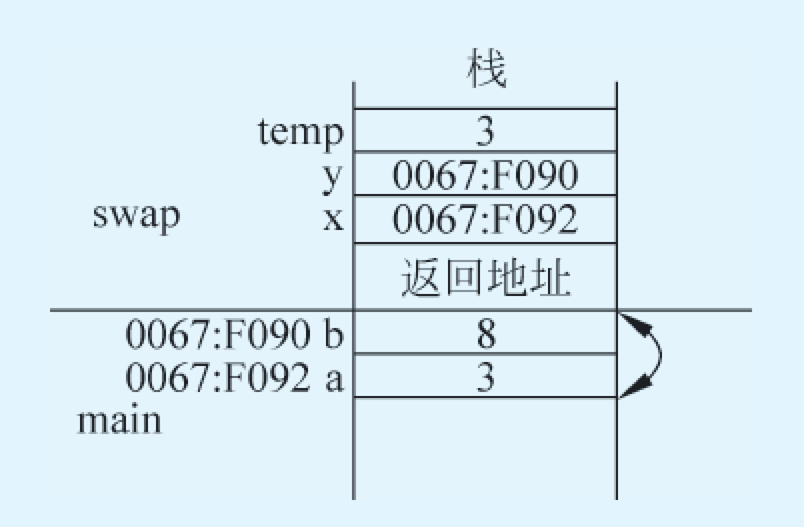


图 8-9 传递指针的内存布局

由于指针可以间接访问变量，使得函数调用中值的返回变得灵活多样，可以更方便地实现函数之间的数据传递。

但是要看到，指针的灵活是以破坏函数的黑盒特性为代价的。它使函数可以访问本函数的栈空间以外的内存区域(函数的副作用初露端倪)，以致引起了以下问题。

(1) 可读性问题：因为间接访问比直接访问相对难理解，传递地址比传递值的直观性要差，函数声明与定义也相对比较复杂。

(2) 重用性问题：函数调用依赖于调用函数或整个外部内存空间的环境，丧失了黑盒的特性，所以无法作为公共的函数模块来使用。

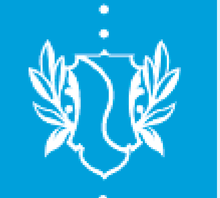
(3) 调试复杂性问题：跟踪错误的区域从函数的局部栈空间扩大到整个内存空间。不但要跟踪变量，还要跟踪地址。错误现象从简单的不能得到相应的返回结果，扩展到系统环境遭破坏甚至死机。

3. 指针函数

返回指针的函数称为指针函数。**指针函数不能把在它内部说明的具有局部作用域的数据地址作为返回值**。

例如，下面的程序打印一个用整型指针指向的整数值：

```
//---------------------
//     ch8_14.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int * getInt(char * str){                //指针函数
   int value = 20;
   cout << str << endl;
   return &value;                        //warning: 将局部变量的地址返回是不妥的
}//---------------------
void somefn(char * str){
   int a = 40;
   cout << str << endl;
}//---------------------
int main(){
```




```
    int * pr = getInt("input a value:"); //赋值取自返回的指针值
    cout << * pr << endl; //第一次输出 * pr
    somefn("It is uncertain.");
    cout << * pr << endl; //第二次输出 * pr
}//--------------------
```

运行结果为：

```
input a value:
20
It is uncertain.
4435500
```

该程序中的 getInt()函数返回一个局部作用域变量的地址是不妥的。因为 getInt()函数结束时，其栈中的变量 value 随之消失。在 BC 编译器中，将给出“可疑的指针转换”(suspicious pointer conversion)警告；在 VC 编译器中将给出“返回了一个局部变量或临时空间的地址”(returning address of local variable or temporary)警告。

在主函数中，指针 pr 得到一个局部变量的地址，输出该地址中的内容 20。随后，调用了另一个函数 somefn()，该函数将前一个被调函数的栈空间位置作为自己的栈工作空间，所以函数返回后，栈中内容发生了变化。而主函数下一个语句输出的是该变化了的内存空间内容。

可以返回堆地址，可以返回全局或静态变量的地址，但不要返回局部变量的地址。

4. void 指针

void 指针是空类型指针，它不指向任何类型，即 void 指针仅仅只是一个地址。所以**空类型指针不能进行指针运算，也不能进行间接引用**，因为指针运算和间接引用都需要指针的类型信息。例如：

```
void * p;      //仅仅表示 p 存放一个地址
p++            //error: +、- 运算离不开指针类型
* p = 20.5;    //error: 访问 p 指向的变量空间需要变量类型信息
```

由于其他指针都包含地址信息，所以将其他指针的值赋给空类型指针是合法的；反之，将空类型指针赋给其他指针则不被允许，除非进行显式转换。例如：

```
int a = 20;
int * pr = &a;
void * p = pr;      //ok: 将整型指针值赋给空类型指针
pr = p;             //error: 不能将空类型指针赋给其他指针
pr = (int * )p;     //ok: 显式转换被允许
```

上面的代码中，将空类型指针赋给其他指针时，在 BC 编译器中将给出“可疑的指针转换”(suspicious pointer conversion)警告；在 VC 编译器中将给出“不能将空类型指针转换成整型指针”(cannot convert from 'void * ' to 'int * ')的编译错误。

下面的程序是空类型指针的一个应用。函数 memcpy()在头文件 string.h 中声明，它的功能为从源 src 中拷贝 n 个字节到目标 dest 中：

```
//-----------------------
//    ch8_15.cpp
//-----------------------
#include <iostream>
#include <string.h>    //用到 memcpy()
using namespace std;
//-----------------------
int main(){
  char src[10] = "*********";
  char dest[10];
  char* pc = (char*)memcpy(dest, src, 10);
  cout << pc << endl;
}//---------------------
```

运行结果为：

```
*********
```

函数 memcpy()的原型为：

```
void* memcpy(void* d,const void* s, size_t n);
```

其中，size_t 为 unsigned int。该函数返回空类型指针 dest 的值，d 形参值是主函数中的实参 dest 的数组首地址。

8.7 字符指针

1. 字符串的表示

在 C++中，字符串是用双引号括起来的字符序列，简称字串，又称字符串字面值。当字符串用于字符数组初始化时，其在完成将内容填写到所创建的字符数组中之后，随即消失，不再另辟存储空间；而当字符串用于表达式，或输出，或赋值，或作参数传递，则其在运行中有它自己的存储空间，可以寻址访问。例如：

```
char buffer[] = "hello";     //字符数组初始化
cout <<"good" << endl;       //字符串用于输出
```

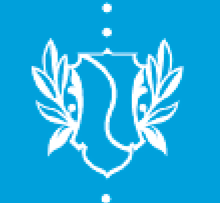
在例中，"hello"用来给字符数组初始化，"good"字符串用来输出。

字符串的类型是指向字符的指针(字符指针 char *)，它与字符数组名同属于一种类型。字符串在内存中以'\0'结尾。这种类型的字符串称为 C 字符串，或 ASCIIZ 字符串(ASCII 序列后跟 Zero 之意)。

2. 字符串的属性

字符串通常存放在内存 data 区中的 const 区，见图 8-10；而字符数组是根据其数据存储的特点存放在相应的位置上。如果字符数组是全局变量，就存放在内存 data 区中的全局或静态区；如果字符数组是局部数据，就存放在内存的栈区等。

当编译器遇到一字符串时，就把它放到字符串池(data 区的 const 区)中，以'\0'作结束符，记下其起始地址，在所构成的代码中使用该地址。这样，字符串就“变成”了地址。



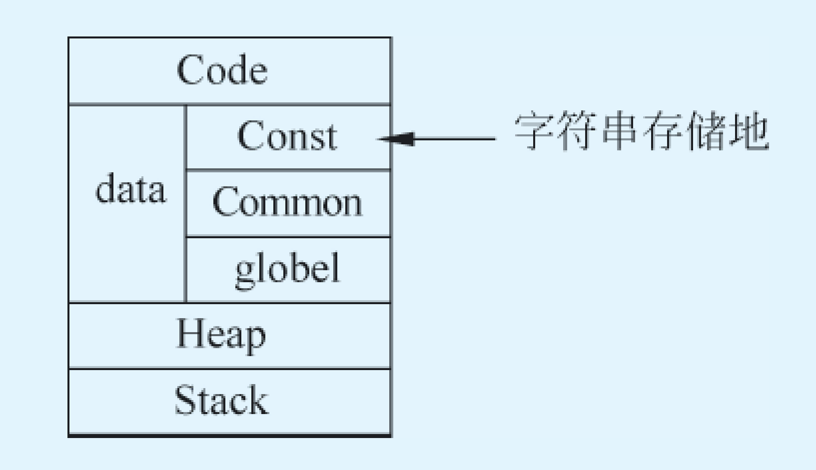


图 8-10　字符串的内存位置

由于字符串的地址属性，所以**两个同样字符组成的字符串的地址是不相等的**。

例如，下面的程序不会输出"equal"字符串：

```
//----------------------
//    ch8_16.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  if("join" == "join")
    cout <<"equal\n";
  else
    cout <<"not equal\n";
}//----------------------
```

运行结果为：

```
not equal
```

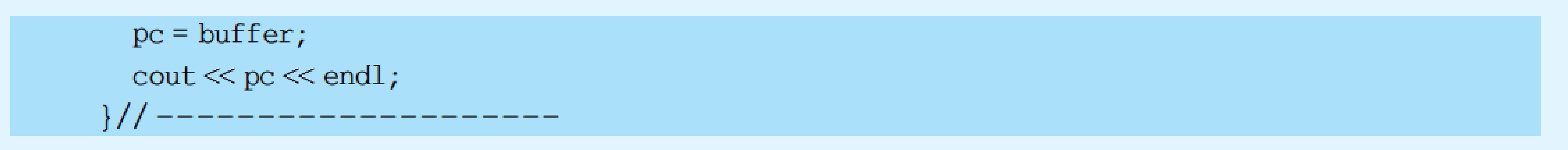
程序中两个字符串的比较实质上是两个地址的比较。在编译时，给了这两个字符串不同的存放地点，所以两个“join”字符串的地址是不同的。要使两个字符串真正从字面上进行比较，可以用库函数 strcmp()，见稍后的描述。

3. 字符指针

字符串、字符数组名、字符指针均属于同一种数据类型。

例如，下面的程序描述了字符指针的操作：

```
//----------------------
//    ch8_17.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  char buffer[10] = "ABC ";
  char * pc;
  pc = "hello";        //ok: 将字符串的首地址赋给指针
  cout << pc << endl;
  pc++;
  cout << pc << endl;
  cout << * pc << endl;
```



```
    pc = buffer;
    cout << pc << endl;
}//---------------------
```

运行结果为：

```
hello
ello
e
ABC
```

buffer 初始化为"ABC"，buffer[3]='\0'，buffer[4]～buffer[9]的内容不重要。

pc 是字符指针，定义时分配该变量空间但没有初始化，之后将"hello"赋给 pc。由于字符串是地址，所以"pc="hello";"语句完全合法。pc 实际上指向"hello"中的'h'字符。当 pc++时，pc 就指向这个字符串中的'e'字符。

输出字符指针就是输出字符串。所以输出 pc 时，便从'e'字符的地址开始，直到遇到'\0'结束。

输出字符指针的间接引用，就是输出单个字符。当输出 * pc 时，便是输出 pc 所指向的字符，见图 8-11。

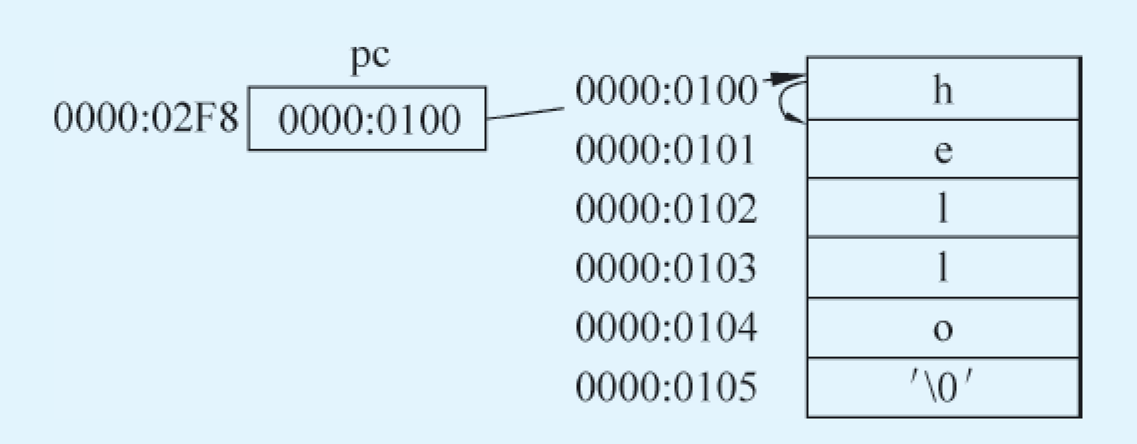


图 8-11　字符指针指向字符串常量

buffer 是字符数组名，所以"pc=buffer;"也是合法的。之后的输出只输出字符数组中的前 3 个字符，遇到'\0'就结束了。

当 pc 指向 buffer 后，字符串字面值"hello"仍逗留在内存的 data 区，但是再也访问不到该字符串(数据丢失)了。所以对于字符串赋给字符指针的情形，指针一般不再重新赋值。

4. 字符串比较

我们已经知道，**两个字符串的比较是地址的比较**。同理，**两个数组名的比较也是地址的比较**。

例如，下面的程序不会输出"equal"：

```
//----------------------
//    ch8_18.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
    char buffer1[10] = "hello";
    char buffer2[10] = "hello";
```




```
    if(buffer1 == buffer2)
      cout <<"equal\n";
    else
      cout <<"not equal\n";
}//---------------------
```

运行结果为:

```
not equal
```

因为 buffer1 和 buffer2 都是地址,buffer1 与 buffer2 是不同的数组,占有不同的内存空间,所以其地址也不同,即 buffer1 与 buffer2 不相等。那么,要进行字符串比较,应该怎么办?

字符串比较应该是逐个字符一一比较,通常使用标准库函数 strcmp(),它在 string.h 或 cstring 头文件中声明,其原型为:

```
int strcmp(const char * str1, const char * str2);
```

其返回值如下:

(1) 当 str1 串等于 str2 串时,返回值 0;

(2) 当 str1 串大于 str2 串时,返回一个正值;

(3) 当 str1 串小于 str2 串时,返回一个负值。

例如,下面的程序修改了程序 ch8_18.cpp,使之成功地进行字符串比较:

```
//----------------------
//    ch8_19.cpp
//----------------------
#include <iostream>
#include <string.h>    //用到 strcmp()
using namespace std;
//----------------------
int main(){
  char buffer1[10] = "hello";
  char buffer2[10] = "hello";

  if(strcmp(buffer1,buffer2) == 0)
    cout <<"equal\n";
  else
    cout <<"not equal\n";
}//---------------------
```

运行结果为:

```
equal
```

5. 字符串赋值

C++中可以用字符串去初始化字符数组,但是不能对字符数组赋予一个字符串,原因是数组名是常量指针,不是左值。

例如,下面的代码不能通过编译:

```
char buffer[10];
buffer = "hello"; //error
```

可以使用标准函数 strcpy()来对字符数组进行赋值。strcpy()的声明在头文件 string.h 中，它的原型为：

```
char * strcpy(char * dest, const char * src);
```

strcpy()函数的返回值为 dest 值，一般都舍弃该值。

例如，下面的代码对字符数组进行赋值：

```
char buffer1[10];
char buffer2[10];
strcpy(buffer1,"hello");
strcpy(buffer2,buffer1);
```

函数 strcpy()仅能对以'\0'作结束符的字符数组进行操作。若要对其他类型的数组赋值，可调用函数 memcpy()：

```
int intarray1[5] = {1,3,5,7,9};
int intarray2[5];
memcpy(intarray2,intarray1,5 * sizeof(int));
```

8.8 指针数组

1. 定义指针数组

一个数组中若每个元素都是一个指针，则为指针数组。

例如，下面的代码定义一个字符指针数组，并对其初始化：

```
char * proname[ ] = { "Fortran",
                      "C ",
                      "C ++ " };
```

该指针数组具有变量属性，所以可以是全局的、静态的和局部的。proname 数组中每个元素都存放一个字符指针(char *)，初始化中的每个值都是一个字符串字面量，这些字面量存储在内存 data 区的 const 子区中，而不管 proname 是全局还是局部变量。字符串在 const 子区中有可能连续分布，也可能不是，但这无关紧要，重要的是指向这些字符串的指针是连续排列在指针数组中的。在 16 位机器中，proname 数组空间占 6 个字节以存放 3 个字符指针，见图 8-12。

2. 指针数组与二维数组

指针数组与二维数组是有区别的。前面看到字符指针数组的内存表示，指针所指向的字符串是不规则长度的。

proname 数组本身含有 6 字节(假设每个指针占 2 字节)；

proname[0]指向的字符串占 8 字节(包括'\0')；

proname[1]指向的字符串占 2 字节；

proname[2]指向的字符串占 4 字节；

proname 所占内存总数为 20 字节。它们分布在不同存储区中。

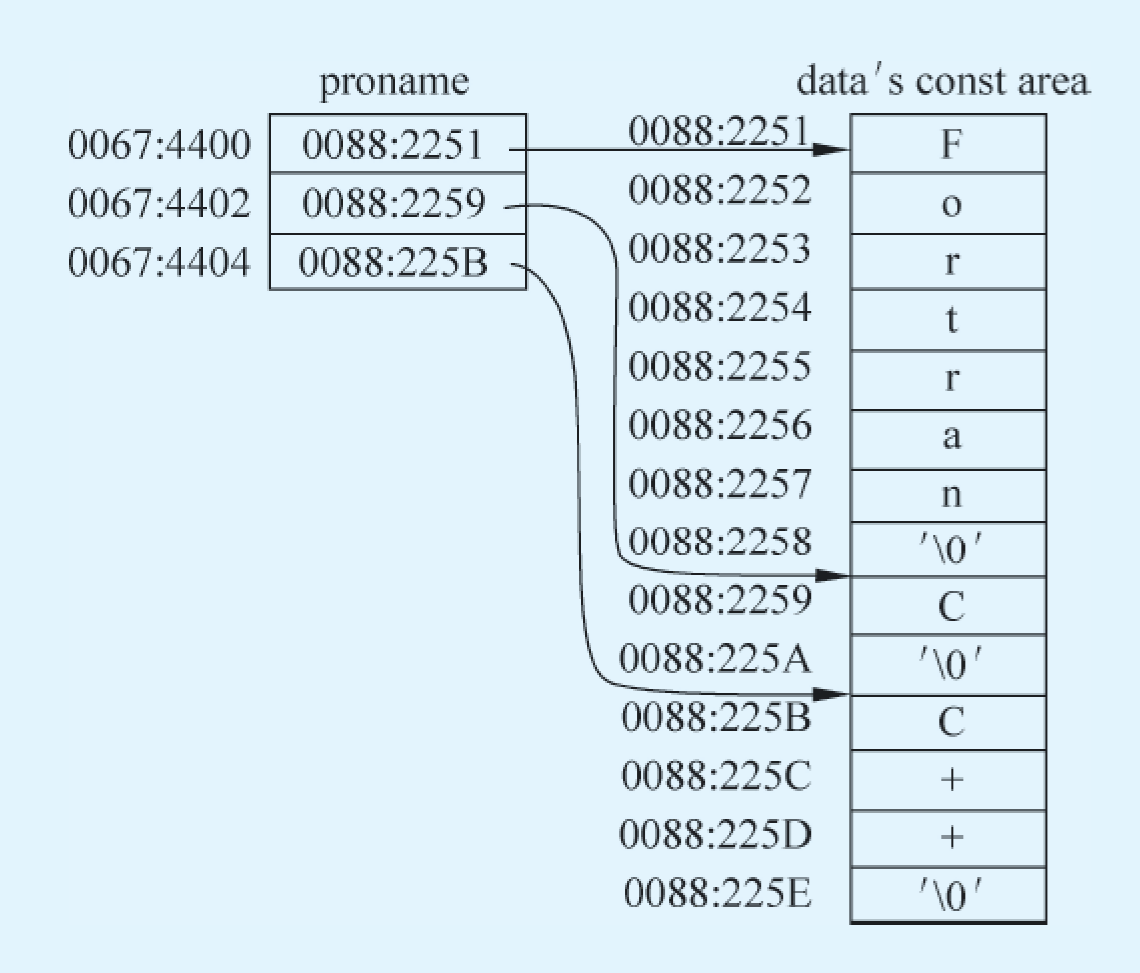


图 8-12　字符指针数组的内存表示

使用二维数组时的情况：

```
char name[3][8] = { "Fortran",
                    "C ",
                    "C ++ "};
```

在二维数组情况下，每一列的大小必须是一样的，因此只能将数组中所要存储的字符串中最长列的大小作为数组的列的大小。这样，共需 3×8＝24 字节。

如果被赋值的各字符串长短相差悬殊，则二维数组空间就浪费多一些。二维数组是作为一个整体存储在某一个区域中。

3. 指向指针的指针

指针数组是数组，则其名字便为指针常量。指针具有类型，那么指针数组名是什么类型呢？

指针数组名是指向指针的指针(即二级指针)。

例如，下面的代码描述了指向字符指针的指针：

```
char * pc[] = {"a","b","c"};
char ** ppc;
ppc = pc;  //ok
```

这里的初始化值"a""b""c"必须为双引号，如果是单引号则表示是字符而不是字符串，而且字符没有'\0'的结束标记。字符总是占一个字节。

传递数组给函数就是传递指针给函数。**传递指针数组给函数就是传递二级指针给函数。**

例如，下面的程序把一个字符指针数组传递给函数：

```
//---------------------
//    ch8_20.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
void print(char * [],int);
```

```
//----------------------
int main(){
  char * pn[] = {"Fred","Barney","Wilma","Betty"};
  int num = sizeof(pn)/sizeof(char * );
  print(pn,num);
}//---------------------
void print(char * arr[], int len){
  for(int i = 0; i < len; i++)      //输出各字符串
    cout <<(int)arr[i]<<" "         //输出字符串地址
        << arr[i]<< endl;           //输出字符串
}//---------------------
```

运行结果为：

```
4202628 Fred
4202633 Barney
4202640 Wilma
4202646 Betty
```

arr 是指向字符指针的指针，所以 * arr 是字符指针，*(arr+i)是以 arr 开始的第 i 个字符指针，*(arr+i)就是 arr[i]。

arr 是指针而不是数组，尽管形参声明为指针数组。这是由传递数组参数即为指针的特性决定的。所以，函数的输出语句可以写为：

```
cout <<(int) * arr;
cout << * arr ++ << endl;      //arr 值可以改变
```

输出字符指针就是输出字符串。如果要输出字符指针的地址值，应该将字符指针强制转换为整型“cout <<(int)arr[i];”。本例中输出的地址是十进制表示的整数，若要以十六进制表示输出，则须用流控制的方法实现。

传递字符指针的指针之内存布局见图 8-13。

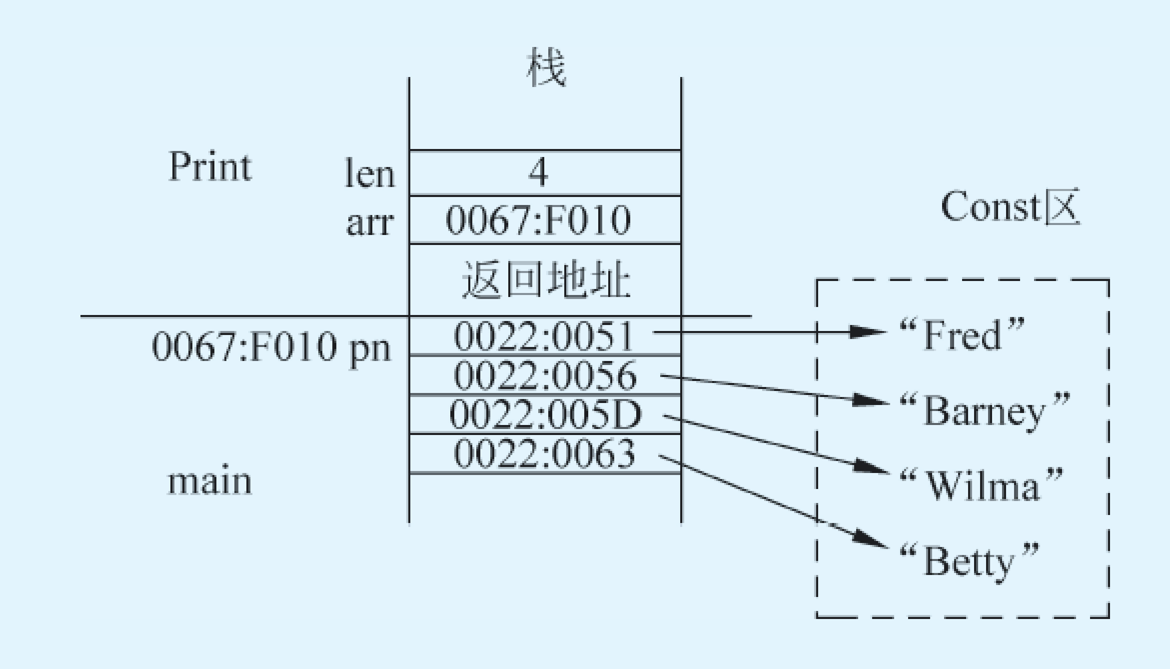


图 8-13　传递字符指针的指针之内存布局

4. NULL 指针值

NULL 是空指针值，它不指向任何地方。不同的操作系统会使编译取不同的 NULL 值。因此，NULL 是个不确定值。在 BC 和 VC 中，NULL 都取 0 值。如果严格考虑到移植，那么不应养成 NULL 即为 0 的习惯。

例如,下面的代码可移植性差:

```
char ch = NULL;      //NULL 是个不确定值
char * pc = NULL;    //ok: 将空指针值赋给 pc
```

在程序中,如果一个数组大小不定,则处理指针数组时可以利用在数组末尾设置 NULL 来解决。

例如,下面的程序通过对 ch8_20.cpp 中的指针数组设置 NULL 的方法进行处理:

```
//----------------------
//     ch8_21.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void print(char * []);
//----------------------
int main(){
  char * pn[] = {"Fred","Barney","Wilma","Betty",NULL};
  print(pn);
}//--------------------
void print(char * arr[]){
  for( ; *arr!= NULL; arr++) //输出各字符串
    cout <<(int) *arr <<" "<< *arr << endl;
}//--------------------
```

运行结果为:

```
4202628   Fred
4202633   Barney
4202640   Wilma
4202646   Betty
```

在主函数传递数组参数给 print()函数时,并不需要将数组元素个数传递过去,因为程序中函数之间达成了一种默契:遇到 NULL 空指针就结束数组处理。

NULL 与 void * 是不同的概念,NULL 是一个值,一个指针值,任何类型的指针都可赋予该值;而 void * 是一种类型,它定义无类型指针。

8.9 命令行参数

1. 命令行参数的概念

C++程序只不过是操作系统调用的函数。

很多程序都需要从命令行输入参数。例如,在 DOS(即 window 下的命令提示符环境)中,copy 命令需要两个参数,type 命令需要一个参数:

```
c> copy filea fileb
c> type c:autoexec.bat
```

操作系统将命令行参数以字符串的形式传递给 main()。因此 main()的第一行的形式应为:

```
int main(int argc, char * argv[])
{
  //...
}
```

这种形式是固定不变的，不能将参数改成其他类型。但由于数组传递的实质是指针，所以第二个参数“char * argv[]”也可表示成“char ** argv”。

整数 argc 和字符指针数组 argv 一直都在栈中，只是以前的：

```
int main()
{
  //...
}
```

形式中，程序忽略了它们。

参数的取名是任意的，可以把 main()的输入参数改成任何喜欢的称呼：

```
int main(int count,char * str[])
```

但是，全世界都已经习惯 argc、argv 这样的称呼，所以习惯成自然了。

2. 打印命令行参数

argc 表示参数个数，argv 表示参数数组。

例如，下面的程序打印命令行参数，从中可看到命令行参数在 argc 和 argv 中的体现：

```
//---------------------
//    ch8_22.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(int argc, char * argv[]){
  int iCount = 0;
  while(iCount < argc){
    cout <<"arg "<< iCount <<": "<< argv[iCount]<< endl;
    iCount++;
  }
}//---------------------
```

运行结果为：

```
C>ch8_22 aBcD eFg hIJkL
arg 0: ch8_22
arg 1: aBcD
arg 2: eFg
arg 3: hIJkL
```

再一次运行结果为：

```
C>ch8_22 c:\file1.img d:\abc\file2.img
arg 0: ch8_22
arg 1: c:\file1.img
arg 2: d:\abc\file2.img
```

参数的个数与内容决定于在程序运行时命令行的表示。argv 是字符指针数组，所以 argv[iCount]是字符指针，指向字符串。命令行参数的栈表示见图 8-14。

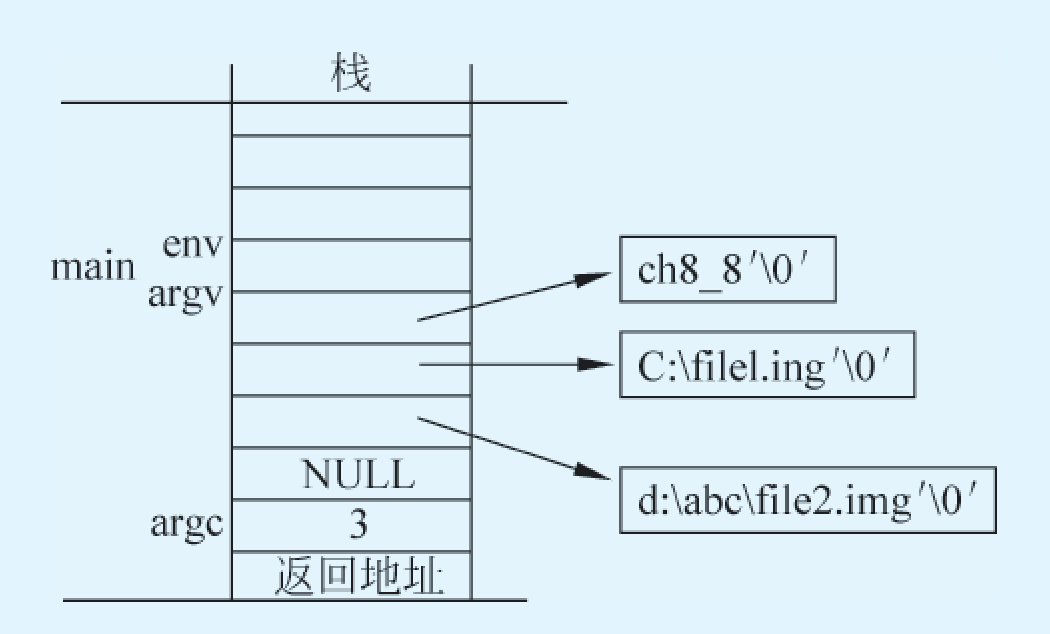


图 8-14　程序的命令行参数栈结构

知道了命令行参数的栈表示，就能更加灵活地运行命令行参数。例如，打印命令行参数的程序，方法不止一种，下面的程序与程序 ch8_22.cpp 的功能一样：

```
//---------------------
//    ch8_23.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(int argc, char* argv[]){
  int i=0;
  while(*argv)
    cout<<"arg "<<i++<<": "<<*argv++<<endl;
}//---------------------
```

3. 命令行参数使用形式

操作系统启动程序时，总是把 3 个参数(int argc，char * argv[]，char * env[])传递给 main()函数，env 用得很少，这里不作介绍。程序可以按下面 4 种方式来声明 main()函数：

- int main()
- int main(int argc)
- int main(int argc，char * argv[])
- int main(int argc，char * argv[]，char * env[])

其中第二种情况是合法的，但不常见，因为在程序中很少有只用 argc，而不用 argv[]的情况。

在命令行参数中，有时某个参数含有空格，而操作系统是以空格作为区分下一个参数的标志。解决的方法是将该参数用引号括起来。

ch8_22.cpp 的程序运行不加引号时，结果为：

```
C>ch8_22 Hello how are you
arg 0: ch8_22
arg 1: Hello
arg 2: how
arg 3: are
arg 4: you
```



ch8_22.cpp 的程序运行加引号时，结果为：

```
C>ch8_22 "Hello how are you"
arg 0: ch8_22
arg 1: Hello how are you
```

4. main()函数的返回

一般来说，main()函数返回程序运行的状态码，例如：

```
int main()
{
   //...
   return 1;
}
```

程序将返回 1。当用户程序使用 exit()子程序终止出口时，它返回用户设定的值。例如：

```
exit(1);
```

返回状态为 1。效果与“return 1”等价。使用 exit()时，可以不论 main()的返回类型。例如：

```
void main()
{
   //...
   exit(1);
}
```

尽管 main()函数返回类型为 void，即无返回值，但仍可使用 exit()给操作系统返回一个值。任何其他的函数使用 exit()，即意味着程序终止，返回到操作系统中。

8.10 函数指针

在程序运行中，全局变量存放在 data 区，局部变量存放在栈区，申请的动态空间存放在堆区。函数代码是程序的算法指令部分，它们同样也占有内存空间，存放在代码(code)区。每个函数都有地址。指向函数地址的指针称为函数指针。函数指针指向代码区中的某个函数，通过函数指针可以调用相应的函数。

1. 定义函数指针

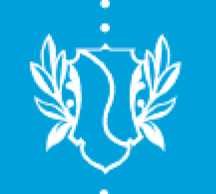
函数指针的定义为：

```
int (*func)(char a, char b);
```

int 为函数的返回类型；被括号括住的 * 和名字表示该名字是一个指针；后面的()表示正在进行函数说明，该指针即是函数指针。该函数具有两个字符型参数 a 和 b。

函数指针有全局、静态和局部之分，它也占有内存空间。与数据指针大小一样，在 16 位机器上占 2 字节，在 32 位机器上占 4 字节。

由于()的优先级大于 *，所以下面是函数定义而不是函数指针定义：




```
int * func(char a, char b);
```

定义中,func 先与()结合构成函数的声明,再得到其返回类型为整型指针 int * 。

2. 函数指针的内在差别

不含下标访问(即方括号[])的数组名是地址,**不作函数调用(即括号())的函数名也是地址**,所以可以将省略了()的函数名作为函数地址赋给函数指针。另一方面,函数是有差别的。不同的参数,不同的返回类型,构成了不同的函数。

例如,下面的代码表示函数和函数指针操作的相互关系:

```
int fn1(char x, char y);        //两个字符参数和返回整型值的函数
int * fn2(char x, char y);      //两个字符参数和返回整型指针的函数
int fn3(int a);                 //一个整型参数和返回整型值的函数

int ( * fp1)(char a, char b);  //两个字符参数和返回整型值的函数指针
int ( * fp2)(int s);           //一个整型参数和返回整型值的函数指针

fp1 = fn1;                      //ok: fn1 函数与指针 fp1 指向的函数一致
fp1 = fn2;                      //error: fn2 函数与 fp1 指向的函数不一致
fp2 = fn3;                      //ok: 函数参数与返回类型相一致,函数名赋给函数指针
fp2 = fp1;                      //error: 两个指针指向的函数不一致
fp2 = fn3(5);                   //error: 函数赋给函数指针时,不能加括号
```

由于函数的差异,导致了函数指针的差异。一个函数不能赋给一个不一致的函数指针。语句"fp1=fn2;"在 BC 编译器中会导致"可疑的指针转换"(suspicious pointer conversion)警告,而在 VC 编译器中会导致"不能将甲函数赋给指向乙函数的指针"的编译错误。两个函数指针的相互赋值同样也要求函数一致。

函数名加上括号就变成了函数调用,所以,语句"fp2=fn3(5);"并不是一个函数地址的赋值,作为函数调用,它返回的是整型数,而 fp2 是一个函数指针,所以会引起数据类型不匹配的编译错误。

函数指针与其他数据类型的指针尽管都是地址,但在类型上有很大的差别。两类指针之间不允许相互赋值,甚至显式转换也不行。因为从意义上来说,函数指针指向程序的 code 区,是程序运行的指令代码,而数据指针指向 data 数据区、stack 栈区和 heap 堆区,是程序赖以运行的各种数据。例如:

```
int * ip;
void ( * fp)();       //函数指针
fp = ip;              //error: 不能相互赋值
ip = fp;              //error: 同上
```

3. 通过函数指针来调用函数

首先必须给函数指针赋值:

```
fp2 = fn3;           //意义见上例
```

然后：

```
fp2(5); 或 (*fp2)(5);
```

第一种方式 fp2(5)是 ANSI C++标准，第二种方式(*fp2)(5)是为了兼容 C 的形式，两种方式都表示同一个函数调用的意义。常用的是第一种方式。

4. 用 typedef 来简化类型名

有时，经常要定义同一种函数指针，最好将该函数指针类型用别名声明下来。以后凡是要用到该函数指针定义，就会方便许多。例如：

```
typedef int(*FUN)(int a, int b);     //声明 FUN 是一个函数指针类型
FUN funp;                            //funp 为一个返回整型和两个整型参数的函数指针
```

FUN 是一个函数指针类型，该指针类型中的指针指向一个函数，它有两个整数参数，返回一个整型数。FUN 不是指针，只是一个指针类型名。通过第二个语句的函数指针定义，才确定一个函数指针 funp。

5. 函数指针用作函数参数

例如，下面的程序计算以 0.10 为步长，特定范围内的三角函数之和：

```
//---------------------
//    ch8_24.cpp
//---------------------
#include<iostream>
#include<cmath>                   //用到 sin()、cos()
using namespace std;
//---------------------
double sigma(double(*func)(double), double dl, double du){
  double dt = 0.0;
  for(double d = dl; d < du; d += 0.1)
    dt  +=  func(d);              //用函数指针调用函数
  return dt;
}//---------------------
int main(){
  double dsum;
  dsum = sigma(sin, 0.1, 1.0);      //sin 函数为实参赋给函数指针 func
  cout <<"the sum of sin from 0.1 to 1.0 is "<< dsum << endl;
  dsum = sigma(cos, 0.5, 3.0);      //cos 函数赋给函数指针 func
  cout <<"the sum of cos from 0.5 to 3.0 is "<< dsum << endl;
}//---------------------
```

运行结果为：

```
the sum of sin from 0.1 to 1.0 is 5.01388
the sum of cos from 0.5 to 3.0 is -2.44645
```

程序中，sigma()函数的第一个参数为函数指针，该指针指向的函数有一个 double 参数并返回 double 类型数。sin 和 cos 就是这样的函数，它们作为实参赋给函数指针 func。其原型在 cmath 的头文件中声明。

又如,标准库函数 qsort()可对任何类型的数组排序。其头文件在 stdlib. h 或 cstdlib 中。qsort()的原型为:

```
void qsort(void * , size_t nelem, size_t width,
           int( * fcmp)(const void * , const void * ));
```

其中,第一个参数为待排序数组; nelem 是数组元素个数; width 是元素类型的长度; fcmp 是函数指针,比较两个参数,如果相等,则返回 0,如果参数 1 大于参数 2,则返回值为正,否则返回值为负。

现排序一个字符串数组,其比较函数用 compare():

```
//---------------------
//    ch8_25.cpp
//---------------------
#include <iostream>
#include <cstdlib>          //用到 qsort()
#include <string.h>         //用到 strcmp()
using namespace std;
//---------------------
int compare(const void* a, const void* b);
//---------------------
char* list[5] = {"cattle","car","cabet","cap","canon"};
//---------------------
int main(){
  qsort((void*)list,5,sizeof(list[0]),compare);

  for(int i = 0; i<5; i++)
    cout << list[i] << endl;
}//--------------------
int compare(const void* a, const void* b){
  return strcmp(*(char**)a, *(char**)b);
}//-------------------- }//--------------------
```

运行结果为:

```
cab
can
cap
car
cat
```

compare()是自定义函数,其比较两个字串的大小,为了要与 qsort()函数中的第四个参数相匹配,所传递的是元素地址,即字串(以地址表达)地址,也即地址的地址,而不是简单的字串地址,所以不能直接用标准库函数 strcmp()必须重写比较函数 compare()。

6. 函数指针可构成指针数组

例如,下面的程序是一个用菜单驱动函数调用的方法,各个函数调用用函数指针数组来实现:

```
//---------------------
//    ch8_26.cpp
//---------------------
#include <iostream>
```

```
using namespace std;
//----------------------
typedef void ( * MenuFun)();     //声明函数指针类型
void f1(){ cout <<"\ngood!\n"; }
void f2(){ cout <<"\nbetter!\n"; }
void f3(){ cout <<"\nbest!\n"; }
int getChoice(){
  cout <<"1-------- display good\n"
      <<"2-------- dispaly better\n"
      <<"3-------- dispaly best\n"
      <<"0-------- exit\n"
      <<"Enter your choice: ";
  int choice;
  cin >> choice;
  return choice;
}//---------------------
MenuFun fun[] = {f1,f2,f3};     //全局函数指针数组
//----------------------
int main(){
  for(int ch; ch = getChoice(); )
    switch(ch){
      case 1: fun[0](); break;
      case 2: fun[1](); break;
      case 3: fun[2](); break;
      default: cout <<"you entered a wrong key.\n";
    }
}//---------------------
```

7. 函数的返回类型可以是函数指针

例如：

```
typedef int ( * SIG)();        //声明返回整型且无参函数的指针类型 SIG
typedef void ( * SIGARG)();    //声明无返回且无参函数的指针类型
SIG signal(int, SIGARG);       //声明返回函数指针的函数
```

最后一行声明一个函数，该函数返回一个函数指针，且有一个整型参数和一个函数指针参数。返回的函数指针是指向返回整型且无参数的函数。作为函数指针参数的指针指向无返回且无参数的函数。

小结

指针不仅仅是地址，还有对数据类型的操作性规定。这是理解指针的关键。

当程序把一个数组传递给函数时，C++实际上把数组第一个元素的地址传递给该函数。通过递增指针的值，可以使指针直接指向数组的下一个元素。

对字符串操作的函数一般用指针遍历字符串，直至指针指向 NULL。当指针指向除字符串以外的其他数组时，函数要知道数组中元素的个数，或者数组中的最后一个元素。

堆允许程序在运行时，而不是在编译时，确定所申请的内存大小。在 C++中，堆分配一般用 new 和 delete 两个操作符，malloc()和 free()函数则相对过时，只在阅读用 C 编制的程



序时才需要这些知识。在第 14 章将进一步讨论 new 和 delete 的使用。

指针使函数的功能大大增强,使之可以访问局部栈以外的内存空间。但也随之带来了副作用,函数的黑盒性受到威胁。为了限制副作用,可以声明函数形参为指针常量和指向常量的指针。

函数指针指向程序的代码区,通过函数指针可以调用函数。程序设计深入下去的时候,必然要触及函数指针,函数指针使程序更加简洁和强大。

指针数组是 C++程序中最有用的结构之一,指针数组和二维数组是有区别的。

命令行参数是让程序员与操作系统之间发生关系,main()是操作系统调用的一个函数。main()从操作系统获取参数值。命令行参数也是指针数组的一个应用。

练习

8.1 下面的程序调用了 findmax()函数,该函数寻找数组中的最大元素,将该元素的下标通过参数返回,并返回其地址值。编程实现 findmax()函数。

```
#include <iostream>
using namespace std;
int * findmax(int * array, int size, int * index);

int main()
{
   int a[10] = {33,91,54,67,82,37,85,63,19,68};
   int * maxaddr;
   int idx;

   maxaddr = findmax(a, sizeof(a)/sizeof( * a), &idx);

   cout <<"the index of maximum element is " << idx << endl
              <<"the address of it is " << maxaddr << endl
              <<"the value of it is " << a[idx] << endl;
}
```

8.2 编写程序,改进 Josephus 问题的解(jose1.cpp),使之在运行时确定小孩数(用 new 和 delete 操作符),并进行输入检查:小孩数不能小于 1,数小孩的间隔数不能小于 1,也不能大于小孩数。发现输入错误应让其选择:停止运行,重输,以默认值 10 个小孩和数小孩间隔 3 让其运行。

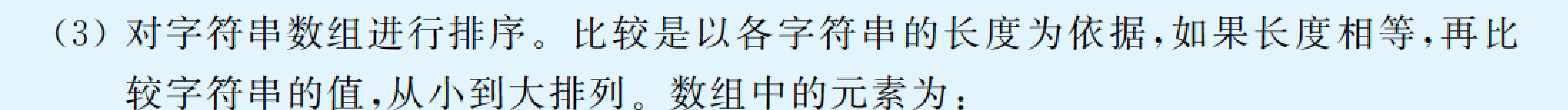
8.3 编写程序,使用标准库函数 qsort(),对各类数组进行排序。

(1) 对整数数组进行排序。比较是以一个整数的各位数字之和的大小为依据,从小到大排列。数组中的元素值为:

```
12, 32, 42, 51, 8, 16, 51, 21, 19, 9
```

(2) 对浮点数组进行排序。从小到大排列。数组中的元素为:

```
32.1, 456.87, 332.67, 442.0, 98.12,
451.79, 340.12, 54.55, 99.87, 72.5
```


(3) 对字符串数组进行排序。比较是以各字符串的长度为依据,如果长度相等,再比较字符串的值,从小到大排列。数组中的元素为:

```
enter, number, size, begin, of, cat, case,
program, certain, a
```

8.4 编写程序,将输入的一行字符加密和解密。加密时,每个字符依次反复加上“4962873”中的数字,如果范围超过 ASCII 码的 032(空格)~122('z'),则进行模运算。解密与加密的顺序相反。编制加密和解密函数,打印各个过程的结果。

例如,加密:the result of 3 and 2 is not 8

```
(t)116 + 4,(h)104 + 9,(e)101 + 6,( )32 + 2,(r)114 + 8,(e)101 + 7,(s)115 + 3,
(u)117 + 4,(l)108 + 9,(t)116 + 6,(o)111 + 2,(f)102 + 8,( )32 + 7,(3)51 + 3,
( )32 + 4,(a)97 + 9,(n)110 + 6,(d)100 + 2,( )32 + 8,(2)50 + 7,( )32 + 3,
(i)105 + 4,(s)115 + 9,( )32 + 6,(n)110 + 2,(o)111 + 8,(t)116 + 7,( )32 + 3,
(8)56 + 4
```

得到密文为:

```
xqk"zlvyuz"wm#7>gpl's$rg" vvw $ A
```

8.5 编写程序,实现两个字符串比较的自定义版:

```
int strcmp(const char * str1,const char * str2);

//当 str1 > str2 时,返回正数
//当 str1 = str2 时,返回 0
//当 str1 < str2 时,返回负数
```

8.6 编写程序,实现复制字符串的自定义版:

```
char * strcpy(char * dest, const char * source);

//该函数返回 dest 的值,即字符串首地址
```

8.7 编写程序,输入命令行参数为两个字符串,用练习 8.5 的 strcmp()比较并输出比较结果。

8.8 编写程序。

(1) 初始化一个矩阵 A(5×5),元素值取自随机函数,并输出;

(2) 将其传递给函数,实现矩阵转置;

(3) 在主函数中输出结果。

随机函数的原型为:

```
int rand();
```

它产生一个 0~65 535 的随机数(16 位机器中)。

8.9 编写程序,测试堆内存的容量:每次申请一个数组,内含 100 个整数,直到分配失败,并打印堆容量报告。

第9章　引用

程序设计语言的进化使用户从被迫解决细节问题中解脱出来，转向花更多时间来考虑“大的蓝图”。根据这种精神，C++中包含了一个称为引用的特性，它允许程序来负责确定把参数传递给函数的方法。学习了本章后，应该掌握引用的语法，用引用传递函数的方法，理解 C++在函数原型中声明引用的目的，正确使用引用，避免不恰当的引用返回，明辨引用与指针的区别。

9.1　引用的概念

引用是个别名，当建立引用时，程序用另一个变量或对象(目标)的名字初始化它。从那时起，引用作为目标的别名而使用，对引用的改动实际就是对目标的改动。

为建立引用，先写上目标的类型，后跟引用运算符“&”，然后是引用的名字。引用能使用任何合法变量名。

例如，引用一个整型变量：

```
int someInt;
int& rInt = someInt;
```

声明 rInt 是对整数的引用，初始化为引用 someInt。在这里，要求 someInt 已经有声明或定义，而引用仅仅是它的别名，不能喧宾夺主。

引用不是值，不占存储空间，声明引用时，目标的存储状态不会改变。所以，既然定义的概念有具体分配空间的含义，那么引用只有声明，没有定义。

例如，下面的程序建立和使用引用：

```
//----------------------
//    ch9_1.cpp
//----------------------
#include <iostream>
using namespace std;
```

```
//---------------------
int main(){
  int intOne = 5;
  int& rInt = intOne;

  cout <<"intOne:"<< intOne << endl;
  cout <<"rInt:"<< rInt << endl;

  rInt = 7;
  cout <<"intOne:"<< intOne << endl;
  cout <<"rInt:"<< rInt << endl;
}//---------------------
```

运行结果为：

```
intOne:5
rInt:5
intOne:7
rInt:7
```

引用 rInt 用 intOne 来初始化。以后，无论改变 intOne 还是 rInt，实际上都是指 intOne，二者的值都一样。

引用在声明时必须被初始化，否则会产生编译错误。

引用运算符与地址操作符使用相同的符号。尽管它们显然是彼此相关的，但它们不一样。

引用运算符只在声明的时候使用，它放在类型名后面，例如：

```
int& rInt = intOne;
```

任何其他"&"的使用都是地址操作符，例如：

```
int * ip = &intOne;
cout << &ip;
```

与指针类似，下面 3 种声明引用的方法都是等价的：

```
int& rInt;
int &rInt;
int & rInt;
```

下面的语句包含一个引用的声明和一个变量的定义：

```
int& rInt, sa;        //会误以为声明了两个引用
```

为了提高可读性，不应在同一行上同时声明引用、指针和变量。

9.2 引用的操作

如果程序寻找引用的地址，它只能找到所引用的目标的地址。

例如，下面的程序取引用的地址：

```
//---------------------
//     ch9_2.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  int intOne = 5;
  int& rInt = intOne;

  cout <<"intOne:"<< intOne << endl;
  cout <<"rInt:"<< rInt << endl;

  cout <<"&intOne:"<< &intOne << endl;
  cout <<"&rInt:"<< &rInt << endl;
}//---------------------
```

运行结果为：

```
intOne:5
rInt:5
&intOne:00F3:5300
&rInt:00F3:5300
```

C++没有提供访问引用本身地址的方法，因为它与指针或其他变量的地址不同，它没有任何意义。引用在建立时就初始化，而且总是作为目标的别名使用，即使在应用地址操作符时也是如此。对引用的理解可以见图 9-1。

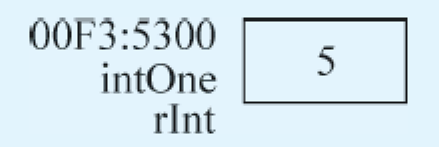


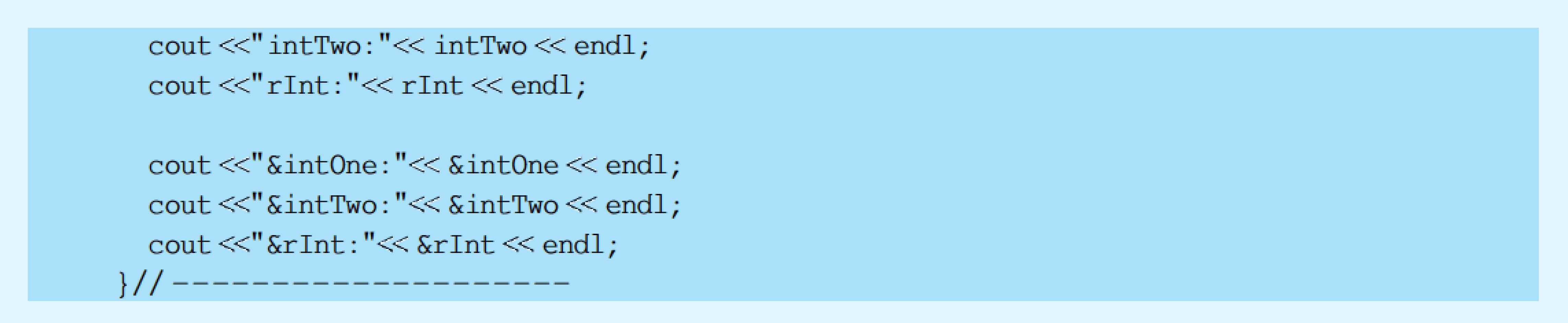
图 9-1　定义 rInt 引用与变量的关系

引用一旦初始化，它就维系在一定的目标上，再也不分开。任何对该引用的赋值，都是对引用所维系的目标赋值，而不是将引用维系到另一个目标上。

例如，下面的程序给引用赋新值：

```
//---------------------
//     ch9_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  int intOne = 5;
  int& rInt = intOne;

  cout <<"intOne:"<< intOne << endl;
  cout <<"rInt:"<< rInt << endl;
  int intTwo = 8;
  rInt = intTwo;
  cout <<"intOne:"<< intOne << endl;
```



```
    cout <<"intTwo:"<< intTwo << endl;
    cout <<"rInt:"<< rInt << endl;

    cout <<"&intOne:"<< &intOne << endl;
    cout <<"&intTwo:"<< &intTwo << endl;
    cout <<"&rInt:"<< &rInt << endl;
}//--------------------
```

运行结果为：

```
intOne:5
rInt:5
intOne:8
intTwo:8
rInt:8
&intOne:0110:F150
&intTwo:0110:F14E
&rInt:0110:F150
```

在程序中，引用 rInt 被重新赋值为变量 intTwo。从运行结果看出，rInt 仍然维系在原 intOne 上，因为 rInt 与 intOne 的地址是一样的，见图 9-2。

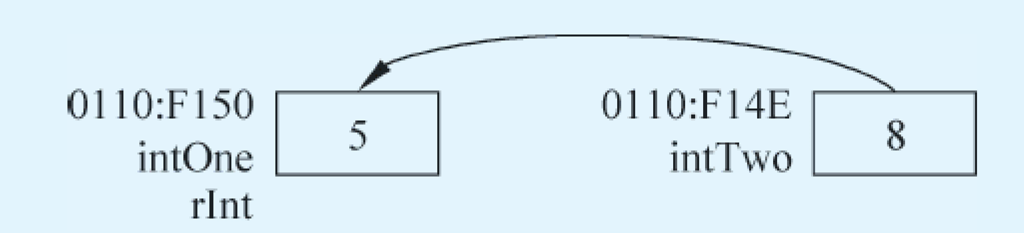


图 9-2　引用被赋值的意义

```
rInt = intTwo;      等价于      intOne = intTwo;
```

引用与指针有很大的差别，指针是个变量，可以把它再赋值成指向别处的地址；然而建立引用时必须进行初始化并且绝不会再指向其他不同的变量。

9.3　什么能被引用

若一个变量声明为 T&，即引用时，它必须用 T 类型的变量或对象进行初始化，或能够转换成 T 类型的对象进行初始化。下面的例子说明整数 1 可以转化成 double 临时变量，从而给引用初始化。

如果引用类型 T 的初始值不是一个左值，那么将建立一个 T 类型的目标并用初始值初始化，那个目标的地址变成引用的值。

例如，下面的代码是合法的：

```
double& rr = 1;
```

在这种情况下：

（1）首先作必要的类型转换；

（2）然后将结果置于临时变量；

（3）最后把临时变量的地址作为初始化的值。

所以上面的语句解释为：

```
double temp;
temp = double(1);
double& rr = temp;
```

上述语句展示了引用容忍非左值初始化的内部处理细节，这只是为兼容函数引用参数传递所面对的非左值实参。但由于这打破了引用对原实体变量访问的本来意义，且在模板编程中，会因为类型匹配问题而遭严格拒绝，所以实际编程中应尽可能避免。

指针也是内存实体，所以可以有指针的引用：

```
int * a;
int * & p = a;      //表示 int * 的引用 p 初始化为 a
int b = 8;
p = &b;             //ok! p 是 a 的别名，是一个指针
```

指针的引用见图 9-3。

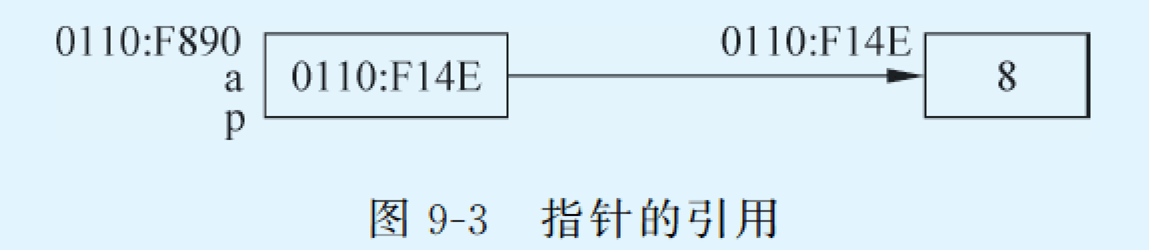


图 9-3　指针的引用

对 void 进行引用是不允许的。例如：

```
void& a = 3;     //error
```

void 只是在语法上相当于一个类型，本质上不是类型，没有任何一个变量或对象，其类型为 void。

不能建立引用的数组：

```
int a[10];
int& ra[10] = a;     //error
```

因为一方面，数组是某个数据类型元素的集合，每个元素皆为引用，意味着每个元素必须初始化为其他内存实体，所以作为引用类型的元素之数组是不现实的；另一方面，数组名只是表示该元素集合空间的起始地址，其代表的是数组整个空间，若对其引用，那就是数组的别名，与指向数组的指针之作用何异？所以数组不该有引用。

引用虽在语法上代表一种类型，但在概念上只是其他实体的附体，所以对同一实体可以定义多个引用，但对不存在的引用实体，就没有引用的引用，也没有指向引用实体的指针（所谓引用的指针）。例如：

```
int a;
int& ra = a;
int&& rra = ra;        //error 无引用的引用
int& * p = &ra;        //error 无引用的指针
```

引用不能用类型来初始化：

```
int& ra = int;     //error
```

因为引用是变量或对象的引用，而不是类型的引用。

有空指针，无空引用。不应有下面的引用声明，否则会有运行错误：



```
int& ri = NULL;     //毫无意义
```

9.4 用引用传递函数参数

1. 引用传递参数

传递引用给函数与传递指针的效果一样，传递的是原来的变量或对象，而不是在函数作用域内建立变量或对象的副本。

在 8.6 节中，我们看到对 swap(int,int)传值方式函数的调用不影响调用函数中的实参，结果并未达到交换数据的预想目的。

使用指针传递方式的 swap(int *, int *)函数的调用，能够达到预定的目的（见 8.6 节），但是函数的语法相对传值方式来说比较累赘。

(1) swap()函数内需要多次显式地间接访问(*pi)，这容易产生错误且难以阅读。

(2) 调用函数需要传递变量地址，使 swap()内部的工作对用户过于透明。

(3) swap(&x,&y)的调用形式会造成一种交换两个变量地址的错觉。

(4) 没有杜绝指针值修改所带来的读写非目标数据的安全隐患。

C++的目标之一就是让使用函数的用户无须考虑函数是如何工作的。传递指针给使用函数的用户增加了编程和理解的负担，这些负担本应属于被调用函数。

例如，下面的程序用引用改写 swap()函数的定义及调用：

```
//----------------------
//    ch9_4.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
void swap(int &x, int &y);
//----------------------
int main(){
  int x = 5, y = 6;
  cout <<"before swap, x:"<< x <<" ,y:"<< y << endl;

  swap(x,y);

  cout <<"after swap, x:"<< x <<" ,y:"<< y << endl;
}//----------------------
void swap(int &rx, int &ry){
  int temp = rx; rx = ry; ry = temp;
}//----------------------
```

运行结果为：

```
before swap, x:5 ,y:6
after swap, x:6 ,y:5
```

在主函数中，调用 swap()函数的参数是 x 和 y，简单地传递变量而不是它们的地址。而事实上，传递的是它们的地址。引用传递的内存布局与指针相仿，只是操作完全不同。每

当使用引用时,C++就去求该引用所含地址中的变量值,见图 9-4。

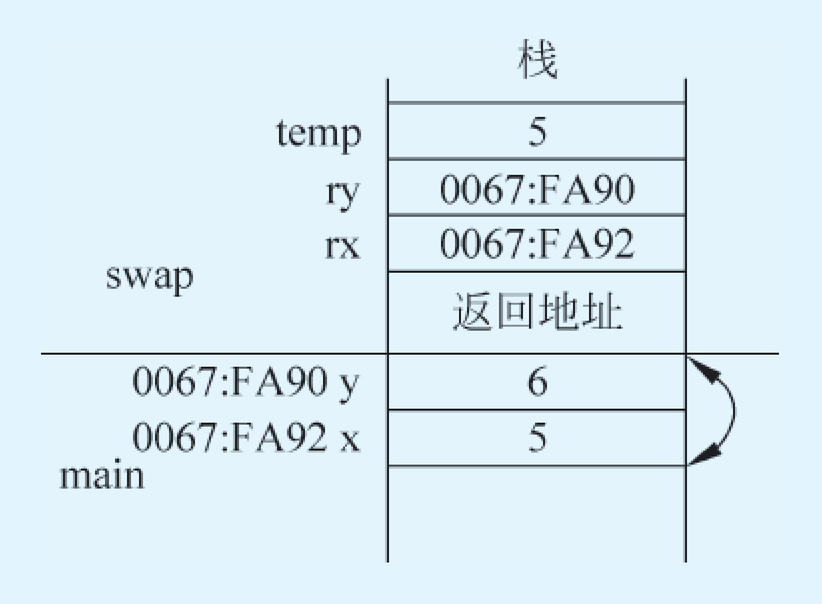


图 9-4　传递引用的内存布局

引用具有指针的威力,但是调用引用传递的函数时,可读性却比指针传递好。引用具有传值方式函数调用语法的简单性与可读性,但是威力却比传值方式强。

2. 引用存在的问题

尽管引用可以表达清晰并让程序员负责了解如何传递参数,但是在有些情况下它们能隐藏错误。

例如,下面的代码在没有看到函数原型之前可能会误认为实参 a 和 b 是通过值来传递的,从而不能通过函数调用来修改它,而事实上却能够修改:

```
int a = 10;
int b = 20;
swap(a,b);
```

因为引用隐藏了函数所使用的参数传递的类型,所以无法从所看到的函数调用判断其是值传递还是引用传递。正因为此,下面的代码中两个重载函数将引起编译报错:

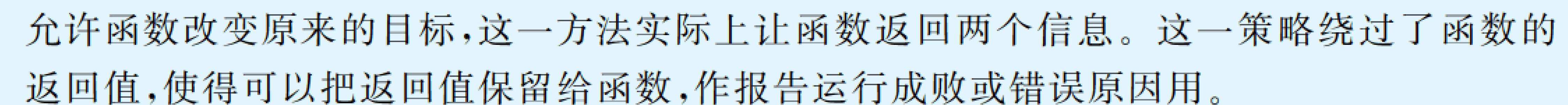
```
void fn(int s)
{
   //...
}

void fn(int& t)
{
   //...
}

int main()
{
   int a = 5;
   fn(a);     //error 匹配哪一个函数?
}
```

9.5　返回多个值

函数只能返回一个值。如果程序需要从函数返回两个值怎么办?解决这一问题的办法之一是用引用给函数传递两个参数,然后由函数往目标中填入正确的值。因为用引用传递



允许函数改变原来的目标,这一方法实际上让函数返回两个信息。这一策略绕过了函数的返回值,使得可以把返回值保留给函数,作报告运行成败或错误原因用。

引用和指针都可以用来实现这一过程。下面的程序实际上返回了3个值,两个是引用,另一个是函数返回值:

```
//---------------------
//    ch9_5.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
bool Factor(int, int&, int&);
//---------------------
int main(){
  int number, squared, cubed;
  cout <<"Enter a number(0~20): ";
  cin >> number;

  bool error = Factor(number, squared, cubed);

  if(error)
    cout <<"Error encountered!\n";
  else{
    cout <<"Number: "<< number << endl;
    cout <<"Squared: "<< squared << endl;
    cout <<"Cubed: "<< cubed << endl;
  }
}//---------------------
bool Factor(int n, int& rSquared, int& rCubed){
  if(n > 20 || n < 0)
    return true;
  rSquared = n * n;
  rCubed = n * n * n;
  return false;
}//---------------------
```

运行结果为:

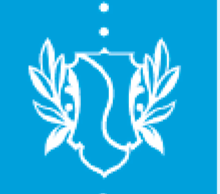
```
Enter a number(0~20): 3
Number: 3
Squared: 9
Cubed: 27
```

Factor()函数检查用值传递的第一参数。如果不在0～20的范围内,它就简单地返回错误值(假设程序正常返回为0)。程序所真正需要的值squared和cubed是通过改变传递给函数的引用返回的,而没有使用函数返回值。

9.6 用引用返回值

函数返回值时,要生成一个值的副本。而用引用返回值时,不生成值的副本。

例如,下面的程序是有关引用返回的4种形式:




```
//----------------------
//    ch9_6.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
float temp;
//----------------------
float fn1(float r){
  temp = r * r * 3.14;
  return temp;
}//---------------------
float& fn2(float r){
  temp = r * r * 3.14;
  return temp;
}//---------------------
int main(){
  float a = fn1(5.0);    //1
  float& b = fn1(5.0);   //2:warning
  float c = fn2(5.0);    //3
  float& d = fn2(5.0);   //4
  cout << a << endl;
  cout << b << endl;
  cout << c << endl;
  cout << d << endl;
}//---------------------
```

运行结果为：

```
78.5
78.5
78.5
78.5
```

对主函数的 4 种引用返回的形式，程序的运行结果是一样的。但是它们在内存中的活动情况是各不相同的。其中，变量 temp 是全局数据，驻留在全局数据区 data。函数 main()、函数 fn1()或函数 fn2()的数据驻留在栈区 stack。

第一种情况：见图 9-5。

这种情况是一般的函数返回值方式。返回全局变量 temp 值时，C++创建临时变量并将 temp 的值 78.5 复制给该临时变量。返回到主函数后，赋值语句 a=fn1(5.0)把临时变量的值 78.5 复制给 a。

第二种情况：见图 9-6。

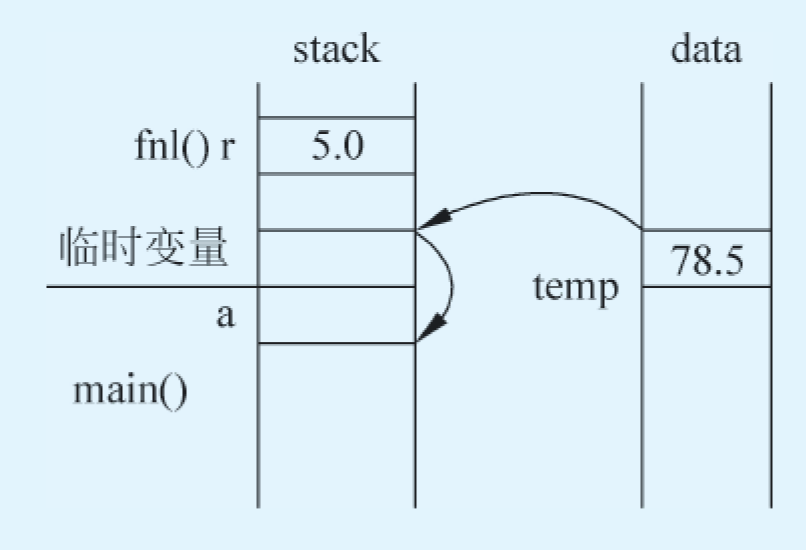


图 9-5　返回值方式的内存布局

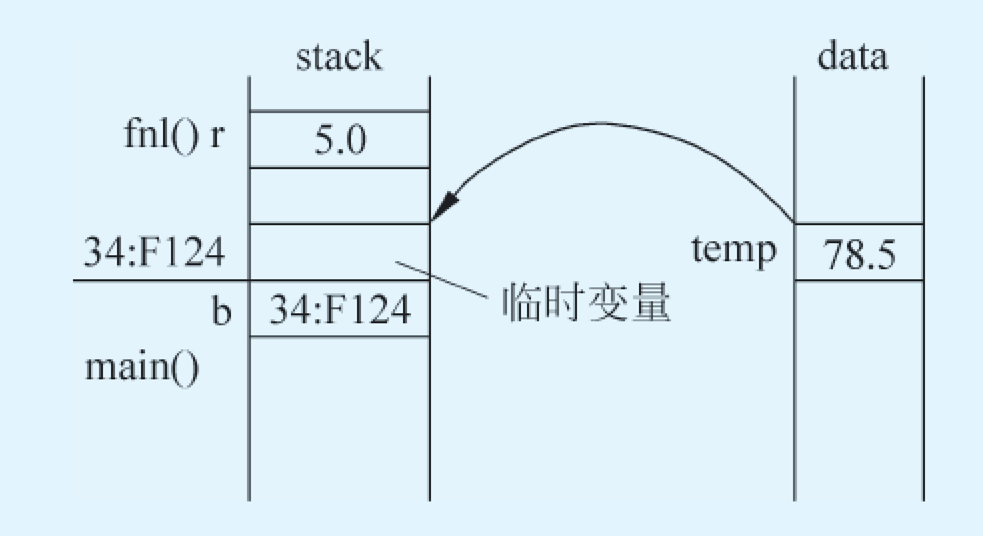


图 9-6　返回值初始引用的情形



这种情况下，函数 fnl()是以值方式返回的，返回时，复制 temp 的值给临时变量。返回到主函数后，引用 b 以该临时变量来初始化，使得 b 成为该临时变量的别名。由于临时变量的作用域随函数返回而终结，所以引用 b 即以非左值初始化规则对待之(见 9.3 节)，赋之以当前 main 函数栈空间的临时变量，以此保证引用 b 在作用域中的依附意义。在 14.7 节中的临时对象的原理也是如此。

若要以返回值初始化一个引用，应该先创建一个变量，将函数返回值赋给这个变量，然后再以该变量来初始化引用，就像下面这样：

```
int x = fn1(5.0);
int& b = x;
```

第三种情况：见图 9-7。

这种情况，函数 fn2()的返回值不产生副本，所以直接将变量 temp 返回给主函数。主函数的赋值语句中的左值 c 直接从变量 temp 中得到复制，这样避免了临时变量的产生。当变量 temp 是一个用户自定义的类型时，这种方式直接带来了程序执行效率和空间利用的利益。

第四种情况：见图 9-8。

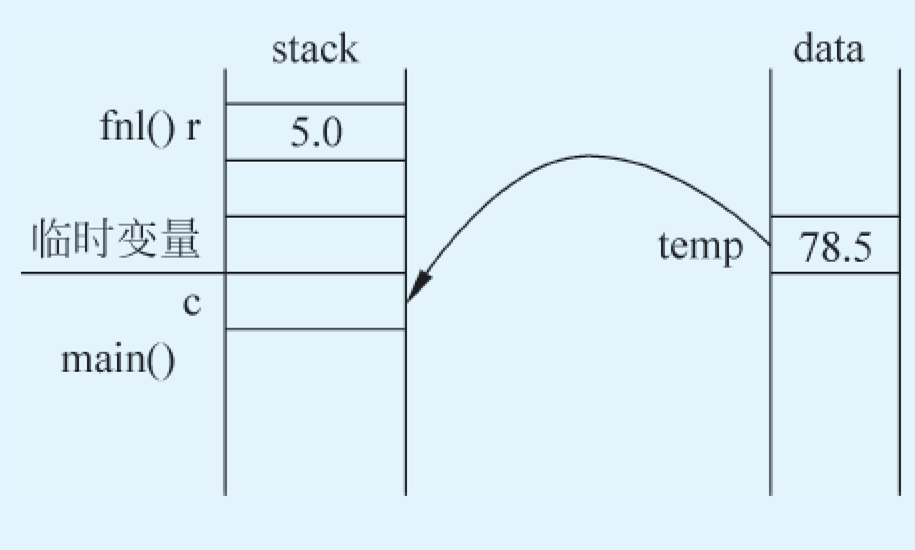



图 9-7　返回引用方式

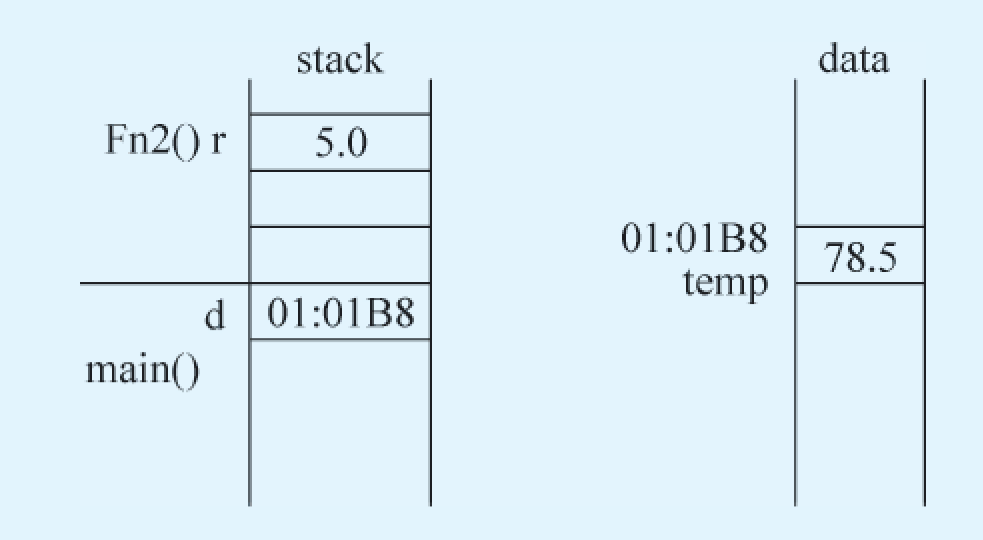


图 9-8　返回引用方式的值作为引用的初始化

这种情况，函数 fn2()返回一个引用，因此不产生任何返回值的副本。在主函数中，一个引用声明 d 用该返回值来初始化，使得 d 成为 temp 的别名。由于 temp 是全局变量，所以在 d 的有效期内 temp 始终保持有效。这样的做法是安全的。

但是，如果返回不在作用域范围内的变量或对象的引用，那就有问题了。这与返回一个局部作用域指针的性质一样严重。BC 作为编译错误，VC 作为警告，来提请编程者注意。

例如，下面的代码返回一个引用，来给主函数的引用声明初始化：

```
float& fn2(float r)
{
   float temp;
   temp = r * r * 3.14;
   return temp;
}

int main()
{
   float& d = fn2(5.0);     //error 返回的引用是个局部变量
}
```

见图 9-9 说明。

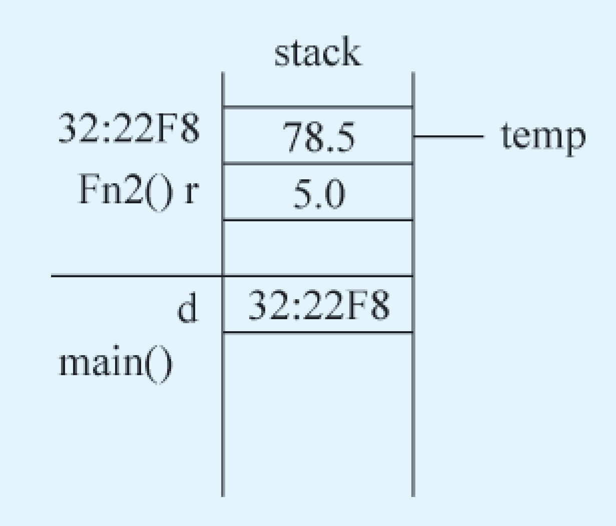


图 9-9　返回的引用是局部变量

如果返回的引用是作为一个左值进行运算，也是程序员最忌讳的。所以，如果程序中有下面的代码，则一定要剔除：

```
float& fn2(float r)
{
   float temp;
   temp = r * r * 3.14;
   return temp;
}

int main()
{
   fn2(5.0) = 12.4;        //error 返回的是局部作用域内的变量
}
```

9.7　函数调用作为左值

在 9.6 节中，对于第三种情况，也意味着**返回一个引用使得一个函数调用表达式成为左值表达式**。只要避免将局部栈中变量的地址返回，就能使函数调用表达式作为左值来使用运行得很好。

例如，下面的程序是统计学生中 A 类学生与 B 类学生各占多少。A 类学生的标准是平均分在 80 分以上，其余都是 B 类学生，先看不返回引用的情况：

```
//---------------------
//    ch9_7.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int array[6][4] = {{60,80,90,75},
                   {75,85,65,77},
                   {80,88,90,98},
                   {89,100,78,81},
                   {62,68,69,75},
                   {85,85,77,91}};
//---------------------
int getLevel(int grade[], int size);
//---------------------
```

```
int main(){
  int typeA = 0,typeB = 0;
  int student = 6;
  int gradesize = 4;

  for(int i = 0; i < student; i++)       //处理所有的学生
    if(getLevel(array[i], gradesize))
      typeA++;
    else
      typeB++;

  cout <<"number of type A is "<< typeA << endl;
  cout <<"number of type B is "<< typeB << endl;
}//---------------------
int getLevel(int grade[], int size){
  int sum = 0;
  for(int i = 0; i < size; i++)          //成绩总分
    sum += grade[i];
  if(sum/size >= 80)                     //平均分大于 80?
    return 1;                            //type A student
  else
    return 0;                            //type B student
}//---------------------
```

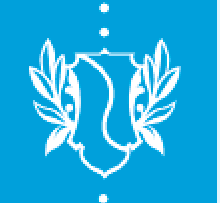
运行结果为：

```
number of type A is 3
number of type B is 3
```

该程序通过函数调用判明该学生成绩属于 A 类还是 B 类，然后给 A 类学生人数增量或给 B 类学生人数增量。

该程序也可以通过返回引用来实现。返回的引用作为左值直接增量。例如：

```
//---------------------
//    ch9_8.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int array[6][4] = {{60,80,90,75},
                   {75,85,65,77},
                   {80,88,90,98},
                   {89,100,78,81},
                   {62,68,69,75},
                   {85,85,77,91}};
//---------------------
int& level(int grade[], int size, int& tA, int& tB);
//---------------------
int main(){
  int typeA = 0,typeB = 0;
  int student = 6;
  int gSize = 4;
  for(int i = 0; i < student; i++)              //处理所有的学生
    level(array[i], gSize, typeA, typeB)++;   //函数调用作为左值
```




```
    cout <<"number of type A is "<< typeA << endl;
    cout <<"number of type B is "<< typeB << endl;
}//--------------------
int& level(int grade[ ], int size, int& tA, int& tB){
    int sum = 0;
    for(int i = 0; i < size; i++)      //成绩总分
        sum += grade[i];

    return (sum/size >= 80 ? tA : tB);
}//--------------------
```

该程序中的 level()函数返回一个引用,为了返回一个非局部变量的引用,就要传递两个引用参数 typeA 和 typeB。当该学生属于 A 类时,就返回 typeA 的引用,否则就返回 typeB 的引用。

由于返回的是引用,所以可以作为左值直接进行增量操作。该函数调用代表 typeA 还是 typeB 的左值视具体的学生成绩统计结果而定。

本例说明:返回引用的函数,可以使函数成为左值。在后面章节中,我们将会看到,这一应用是很多的,最典型的是 cout 和 cin 的操作符重载。

→ 上面各个图中,对引用的表示依赖于实现。引用变量概念上是不占空间的,引用变量被理解为粘附在初始化的实体上,它的实现对用户来说不可见。但并不等于具体实现的时候,非得不占任何空间。从引用变量的初始化,引用变量的访问,到引用参数传递和返回,为了帮助理解引用,让读者有个引用实现的感性认识,这里只是给出了一种实现引用的"指针"方案,图中引用实体用来存放所代表变量的地址。

9.8 用 const 限定引用

传递指针和引用更大的目的是效率。当一个数据类型很大(后面章节中介绍的自定义类型)时,因为传值要复制副本,所以不可取。

另外,传递指针和引用存在传值所没有的危险。程序有时候不允许传递的指针所指向的值被修改或者传递的引用被修改,但传递的地址特征使得所传的参数处于随时被修改的危险之中。

保护实参不被修改的办法是传递 const 指针和引用。

例如,下面的程序传递一个 const double 型的常量指针,返回一个指针:

```
//---------------------
//    ch9_9.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
double * fn(const double * pd){
    static double ad = 32;
    ad += * pd;
    cout <<"fn being called...the value is: "<< * pd << endl;
    return &ad;
}//--------------------
int main(){
```

```
    double a = 345.6;
    const double * pa = fn(&a);
    cout << * pa << endl;
    a = 55.5;
    pa = fn(&a);
    cout << * pa << endl;
}//--------------------
```

运行结果为：

```
fn being called...the value is: 345.6
377.6
fn being called...the value is: 55.5
433.1
```

程序中 fn()函数声明的参数为 double 型的常量指针，返回 double 型的指针。函数 fn()中，没有生成实参 a 的副本，访问 * pd 就是直接访问 a。尽管 a 是变量，但由于限定了 pd 的性质，所以在 fn()函数的作用域范围内，pd 只能以 * pd 的形式读出 a，而不能修改 a。

函数 fn()中定义了一个静态局部变量 ad。在返回时，将其地址返回给了主函数中的常量 double 的指针 pa，使之通过 * pa 能读出 ad 的值而不能修改之。但在函数 fn()中，ad 是变量，是可以被修改的。

将上面的程序改成传递引用与返回引用，函数处理起来会更容易，可读性也更好：

```
//---------------------
//    ch9_10.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
double& fn(const double& pd){
    static double ad = 32;
    ad += pd;
    cout <<"fn being called...the value is: "<< pd << endl;
    return ad;
}//--------------------
int main(){
    double a = 345.6;
    double& pa = fn(a);
    cout << pa << endl;
    a = 55.5;
    pa = fn(a);
    cout << pa << endl;
}//--------------------
```

运行结果为：

```
fn being called...the value is: 345.6
377.6
fn being called...the value is: 55.5
433.1
```

程序 ch9_10.cpp 与 ch9_9.cpp 的输出是一样的。唯一明显的区别是现在的函数 fn()以 const double 的引用为参数，返回 double 的引用。用引用比用指针更简单些，而且程序

可以达到相同的效率，也具有使用 const 所提供的阻断写操作的安全性。

→ C++不区分变量的 const 引用和 const 变量的引用。程序绝不能给引用本身重新赋值，使它指向另一个变量，因此引用总是 const 的。如果对引用应用关键词 const，其作用就是使目标成为 const 变量。即没有：

```
const double const& a = 1;
```

只有：

```
const double& a = 1;
```

9.9 返回堆中变量的引用

对引用的初始化，可以是变量，可以是常量，也可以是一定类型的堆空间变量。但是，由于引用不是指针，所以下面的代码直接从堆中获得的变量空间来初始化引用是错的：

```
int& a = new int(2);     //a 不是指针
```

考虑操作符 new。如果 new 不能在堆空间成功地获得内存分配，它返回 NULL。因为引用不能是 NULL，在程序确认它不是 NULL 之前，程序不能用这一内存初始化引用。

例如，下面的代码说明如何处理这一校验：

```
#include <iostream>
using namespace std;
void fn()
{
   int * pInt = new int;
   if(pInt == NULL)
   {
     cout <<"error memory allocation!";
     return 1;
   }
   int& rInt = * pInt;
   //...
}
```

int 指针 pInt 获得 new 返回的值，程序测试 pInt 中的地址，如果它是 NULL，则报告错误信息并返回；如果它不是 NULL，则将 * pInt 初始化引用 rInt。如此，rInt 成为 new 返回的 int 的别名。

用堆空间来初始化引用，要求该引用在适当时候释放堆空间。

例如，下面的程序在堆中分配空间，求值，然后释放堆空间：

```
//---------------------
//     ch9_11.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
bool circleArea(){
```

```
    double * pd = new double;
    if(!pd){
      cout <<"error memory allocation!";
      return true;
    }
    double& rd = * pd;
    cout <<"the radius is: ";
    cin >> rd;
    cout <<"the area of circle is "<< rd * rd * 3.14 << endl;
    delete &rd;
    return false;
  }//--------------------
  int main(){
    if(circleArea())
      cout <<"program failed.\n";
    else
      cout <<"program successed.\n";
  }//--------------------
```

运行结果为：

```
the radius is: 12
the area of circle is 452.16
program successed.
```

在计算圆面积的 circleArea()中，double 指针接受 new 返回的堆空间地址，然后进行有效性校验。如果有效，则将 * pd 初始化引用 rd，rd 接受键盘输入，计算，打印输出圆面积，返还堆空间，并正常返回；否则输出错误信息，返回出错标志。

这里，返还堆空间有两种方式，一种是 delete pd，另一种是 delete &rd，因为 &rd 和 pd 都指向同一个堆空间地址。对引用来说，同样存在由一个函数建立的堆内存由另一个函数释放的问题，我们在第 8 章有过类似的说明。

对使用堆的引用，有下面的经验：

- 必要时用值传递参数；
- 必要时返回值；
- 不要返回有可能退出作用域的引用；
- 不要引用空目标。

→ *引用和指针使函数的"黑盒"性被打破。函数可以访问不属于自己栈空间的内存。这对把握不住 C++ 的人来说是危险的，而对熟练的程序员来说，正是引用和指针才使函数只能返回单一值的状态被打破，使得函数功能更趋强大。函数的副作用是良性还是恶性，各人自有评说，对于函数潜在的破坏性，是放任自流，还是适当地抑制，专家们动尽了脑筋，直到 C++ 类机制的实现，才使函数的恶性作用得以控制。*

小结

引用是 C++ 独有的特性。指针存在种种问题，间接引用指针会使代码可读性差，易编程出错。而引用正好扬弃了指针。因为引用行使了指针之间接访问的功用，却把指针修改的



操作限定在初始化环节,从而避免了大部分因修改指针而引起的隐性错误。而限定引用必须初始化,又引来了引用操作的简单性。引用的使用中,单纯取个别名是毫无意义的,引用的目的主要用于在函数参数传递中,解决大对象的传递效率和空间都不如意的问题。本章主要介绍引用的原理和各种语法现象,未涉及大对象,它在以后各章陆续介绍。引用能够保证参数传递中不产生副本,从而发挥指针的威力,提高传递的效率,通过 const 的使用,保证了引用传递的安全性。

引用是 C++语言学习中的一个难点。有的人 C 的背景知识不是很强,他们经常一开始不管在什么地方都使用引用,除非有的地方必须使用指针。而另一些学习 C++的 C 程序员则通常避免使用引用,只是把它们作为另一种传递地址的方法来考虑。由于指针也能做这项工作,所以他们只使用指针。这都是片面的。

引用具有表达清晰的优点。引用将对传递的参数的责任附给了编写函数的程序员,而不是使用它们的各个用户。引用是对操作符重载必不可少的补充(见 18.4 节)。没有引用描述形式,所要重载的操作符作为左值的操作数将难以表达。++X 不至于写成++(&x)。引用通过传递地址提高了函数运行的效率。引用传递与值传递在使用方法上比较,唯一的区别是函数的形式参数声明。

不允许声明引用数组,可以用常量来初始化引用声明。返回引用时,要注意局部对象返回的危险。要注意引用隐藏函数所使用的参数传递的类型。

练 习

9.1 读下列程序。

(1) 将其改写为传递引用参数;

(2) 说出其功能;

(3) 将 findmax()函数改写为非递归函数(重新考虑参数个数)。

```
#include <iostream>
using namespace std;
const size = 10;

void findmax(int * a, int n, int i, int * pk);

int main()
{
   int a[size];
   int n = 0;
   cout <<"please input " << size <<"datas:\n";
   for(int i = 0; i < size; i ++ )
   {
     cin >> a[i];
   }

   findmax(a, size, 0, &n);

   cout <<"the maximum is " << a[n] << endl
         <<"It's index is " << n << endl;
}
```

```
void findmax(int * a, int n, int i, int * pk)
{
   if(i<n)
   {
     if(a[i]>a[ * pk])
    * pk = i;
     findmax(a,n,i+1,&( * pk));
   }
}
```

9.2 读下列程序，该程序生成有 10 个整数的安全数组。要把值放入数组中，使用 put()函数，然后取出该值，使用 get()函数，put()和 get()中若遇下标越界立即终止程序运行。其运行结果如后面所示，请完成两个未写出的函数定义。

```
#include <iostream>
using namespace std;
int& put(int n);        //put value into the array
int get(int n);         //obtain a value from the array

int vals[10];
int error = -1;

int main()
{
   put(0) = 10;         //put values into the array
   put(1) = 20;
   put(9) = 30;

   cout << get(0) << endl;
   cout << get(1) << endl;
   cout << get(9) << endl;

   put(12) = 1;         //out of range

}

//函数定义
```

运行结果为：

```
10
20
30
range error in put() value!
```

9.3 编制程序，调用传递引用的参数，实现两个字符串变量的交换。例如开始：

```
char * ap = "hello";
char * bp = "how are you?";
```

交换的结果使得 ap 和 bp 指向的内容分别为：

```
ap:     "how are you?"
bp:     "hello"
```

第10章　结构

到目前为止，我们所见到的数据类型都只包含一种类型信息，即使是多个元素的数组。但是，信息的逻辑关系要求各个数据类型组合在一起考虑会更加方便。结构就能表达这种数据聚集。另外，结构是实现链表结构的理想表现手段。学习本章后，应能掌握结构声明、结构变量定义与访问结构成员的方法，掌握结构作为参数传递与返回结构的函数方法，掌握链表结构的各项基本操作。

10.1　结构概述

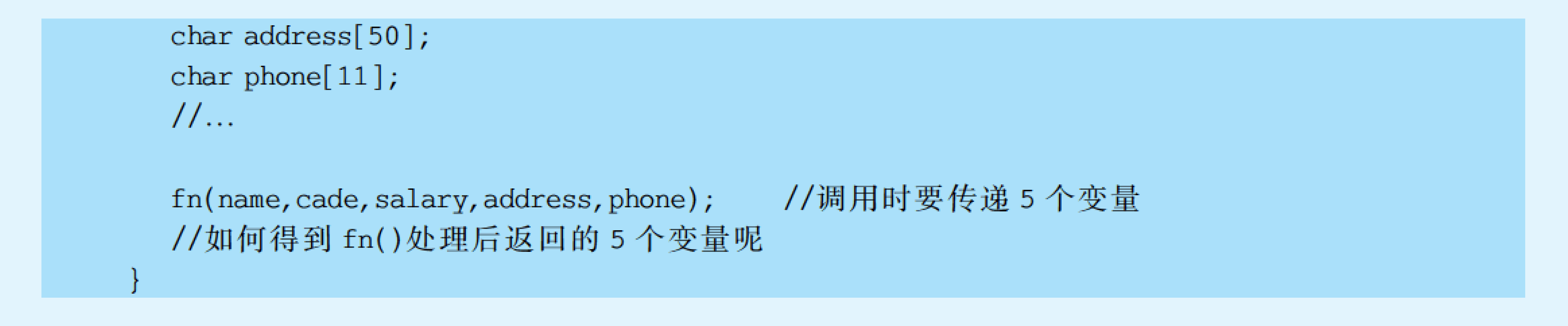
1. 为什么要用结构

C++用数组存储许多相同类型和意义的相关信息，但是有些数据信息是由若干不同数据类型和不同意义的数据所组成。例如，一个人事记录包括姓名、职工编号、工资、地址、电话等，这些数据信息的类型是不一样的，不能用数组的形式把它们组织起来。用结构变量就可以有组织地把这些不同类型的数据信息存放在一起。否则程序需要若干个不同数据类型的变量分别存储职工的信息，不便于程序管理。

例如，定义了一个职工的若干数据后，在函数参数传递时感到麻烦，在返回一个职工信息时遇到了困难：

```
void fn(char * ,long,float,char * ,char * )
{
   //处理职工数据
   //...
   //不知如何返回这 5 个数据
}

int main()
   char name[20];     //定义一个职工要下列 5 个变量定义语句
   long code;
   float salary;{
```



```
    char address[50];
    char phone[11];
    //...

    fn(name,cade,salary,address,phone);     //调用时要传递 5 个变量
    //如何得到 fn()处理后返回的 5 个变量呢
}
```

2. 结构的概念

结构是用户自定义的新数据类型，除此之外，它可与 int、float 等基本数据类型同等看待。声明结构类型时，首先指定关键字 struct 和结构名，然后用一对大括号将若干个结构成员数据类型说明括起来。

通常情况下，结构声明在所有函数之外，位于 main()函数之前。这使新声明的数据类型在程序的任何地方都可以被使用。

例如，声明一个职工 Employee 结构数据类型，它包括姓名、职工编号、工资、地址、电话。用一个结构数据类型的变量可以存放所有这些相关的信息：

```
struct Employee         //名为 Employee 的结构声明
{
    char name[20];
    long code;
    float salary;
    char address[50];
    char phone[11];
};                      //分号是必需的

int main()
{
    Employee person;    //定义一个 Employee 结构的变量，分配变量空间
    //使用这个结构变量
}
```

声明一个结构并不分配内存，内存分配发生在定义这个新数据类型的变量中。

结构中包含的数据变量称为该结构的成员，如 code、salary 是结构 Employee 的成员。

3. 访问结构成员

一旦通过定义相应结构变量，分配了空间，就可以使用点操作符“.”（或称结构成员操作符）来访问结构中的成员。左操作数为结构类型变量，右操作数为结构中的成员。点操作符运算的结果可以是左值，也可以是右值。

在数组中，我们称数组分量为元素；在结构中，我们称结构分量为成员。数组的[]运算符与结构的点运算符具有相同的运算优先级，它们是所有运算符中优先级最高的。

例如，下面的程序说明了访问结构成员的方法：

```
//----------------------
//     ch10_1.cpp
//----------------------
#include <iostream>
```




```
using namespace std;
//----------------------
struct Weather{
  double temp;                    //温度
  double wind;                    //风力
}; //--------------------
int main(){
  Weather today;
  today.temp = 30.5;              //作为左值使用
  today.wind = 10.1;

  cout <<"Temp = "<< today.temp << endl; //作为右值使用
  cout <<"Wind = "<< today.wind << endl;
}//--------------------
```

运行结果为：

```
Temp = 30.5
Wind = 10.1
```

程序中 today.temp 被赋值为 30.5，today.wind 被赋值为 10.1。在点操作符的左右两边都没有空格，这不是必需的，只是一般的程序书写风格。即也可以写成下面这样：

```
today  .  wind = 10.1;
```

在 Weather 结构中，含有两个 double 类型的成员 temp 与 wind，所有成员的数据类型相同，为什么不用数组来实现呢？即为什么不写成：

```
double Weather[2] = {30.5, 10.1};
```

因为我们用结构来实现的目的，就是要区分一个数据聚集中的每个分量的类型和意义，每个分量数据类型可以相异，更重要的是，为了区分数据内容的意义，结构的成员有自己单独的名字，也就能区分每个成员各自的意义。

4. 给结构赋值

数组是不能彼此赋值的。

例如，下面数组的赋值语句会导致一个编译错误：

```
void f()
{
   char a[10],b[10];
   a = b;     //error
   //...
}
```

这主要是因为数组名在性质上是一个常量指针，不允许被赋值。**数组是单纯空间意义上同类数据实体的集合**，数组只是空间的代表，可用数组下标[]操作对单个元素进行访问，但不能对数组(空间地址)作批量元素操作，如 a＝b。其无法看作是同类型数据之间的赋值。

结构就不同了，它大小固定，可以被赋值。

例如，下面的程序对结构变量赋值：

```
//----------------------
//    ch10_2.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
Person pr1 = {"Frank Voltaire", 12345678, 3.35};
//----------------------
int main(){
  Person pr2;
  pr2 = pr1;      //结构变量的赋值
  cout << pr2.name << " "
      << pr2.id << " "
      << pr2.salary << endl;
}//---------------------
```

运行结果为：

```
Frank Voltaire    12345678    3.35
```

程序中定义了一个全局 Person 结构变量 pr1，它使用与初始化数组相似的方法进行初始化。在 main()函数中，定义了一个结构变量 pr2，然后使用赋值运算符将 pr1 的内容赋值给 pr2。

在结构 Person 中，成员 name 是一个字符数组，通过结构变量的赋值，该数组作为成员也被赋值了。

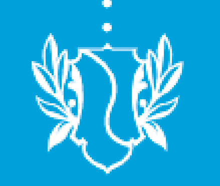
两个不同结构名的变量是不允许相互赋值的，即使二者包含同样的成员。

例如，下面的代码不能将结构 Employee 的变量赋给结构 Person 的变量：

```
struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

struct Employee     //Employee 与 Person 具有相同的成员
{
   char name[20];
   unsigned long id;
   float salary;
};

int main()
{
   Person pr1 = {"Frank Voltaire",12345678,3.35};
   Employee er1;
   er1 = pr1;       //error 类型不匹配
   //...
}
```



→ 在 C 中(而不是 C++),结构变量定义在结构类型名前必须有 struct 关键字。例如,定义结构变量 pr1:

```
struct Person pr1 = {"Frank Voltaire",12345678,3.35};
```

否则 C 编译器报错。

10.2 结构与指针

根据结构类型可以定义一个变量,**是变量就有地址**。结构不像数组,**结构变量不是指针**。通过取地址"&"操作,可以得到结构变量的地址,这个地址就是结构的第一个成员地址。

可以将结构变量的地址赋给结构指针,结构指针通过箭头操作符"->"(也是一种结构成员操作符)来访问结构成员。

例如,下面的代码定义了结构指针,通过结构指针来访问结构成员:

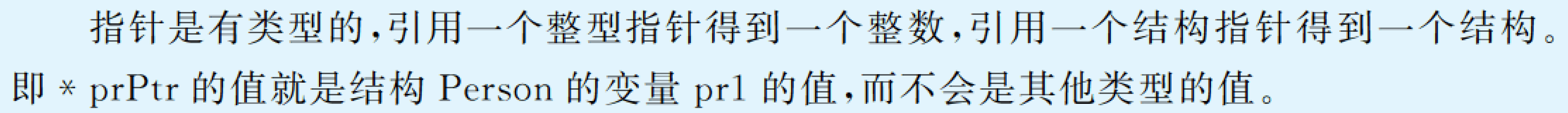
```
//----------------------
//    ch10_3.cpp
//----------------------
#include <iostream>
#include <string>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
int main(){
  Person pr1;
  Person* prPtr;
  prPtr = &pr1;
  strcpy(prPtr->name, "David Marat");
  prPtr->id = 987654321;
  prPtr->salary = 335.0;
  cout << prPtr->name << " "
      << prPtr->id << " "
      << prPtr->salary << endl;
}//--------------------
```

运行结果为:

```
David Marat   987654321   335
```

使用箭头操作符就是对结构成员进行操作。但必须清楚,**当用点操作符时,它的左边应是一个结构变量;当用箭头操作符时,它的左边应是一个结构指针。**

本例中把 pr1 的地址赋给 Person 结构指针 prPtr,然后通过这个指向 pr1 的指针对 pr1 进行赋初值和输出。这一步是重要的,如果不把 pr1 的地址赋给 prPtr,那么 prPtr 是个随机地址,在这个地址上赋值是危险的。



指针是有类型的，引用一个整型指针得到一个整数，引用一个结构指针得到一个结构。即 * prPtr 的值就是结构 Person 的变量 pr1 的值，而不会是其他类型的值。

箭头操作符与点操作符是可以互换的，例如在程序 ch10_3.cpp 中：

```
prPtr -> name    等价于    pr1.name    等价于    ( * prPtr).name
```

也就是说，可以使用点操作符而不使用箭头操作符。

例如，下面的程序将程序 ch10_3.cpp 中结构的箭头操作符改为结构指针：

```
//----------------------
//     ch10_4.cpp
//----------------------
#include <iostream>
#include <string>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
int main(){
  Person pr1;
  Person * prPtr;
  prPtr = &pr1;
  strcpy(( * prPtr).name, "David Marat");
  ( * prPtr).id = 987654321;
  ( * prPtr).salary = 335.0;
  cout <<( * prPtr).name <<" "
       <<( * prPtr).id <<" "
       <<( * prPtr).salary << endl;
}//---------------------
```

运行结果为：

```
David Marat   987654321   335
```

程序的运行结果与 ch10_3.cpp 一样，但这个句法没有真正的好处，通常在这种情况时，用箭头操作符更直观一些。

10.3 结构与数组

结构是一个数据类型，所以也可以拥有结构数组。要定义结构数组，必须先声明一个结构，然后定义这个结构类型的数组。

例如，定义一个 100 个元素组成的 Person 结构类型数组：

```
struct Person
{
   char name[20];
   unsigned long id;
   float salary;
```




```
};

Person allone[100];    //定义一个 Person 类型的数组
```

结构数组中,每个元素都是结构变量,访问结构数组元素中的成员,方法与前类似。例如,下面的程序中,对一个 Person 结构数组进行冒泡法排序,工资高的排在后面:

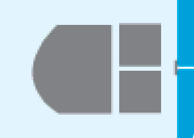
```
//---------------------
//    ch10_5.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
Person allone[6] = {{"jone", 12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
                    {"jasen", 42876, 623.0},
                    {"peter", 23987, 400.0},
                    {"yoke", 12335, 511.0}};
//---------------------
int main(){
  for(int i = 1; i < 6; i++)                     //排序
  for(int j = 0; j <= 5 - i; j++)                //一轮比较
    if(allone[j].salary > allone[j + 1].salary)  //比较工资成员
    {
      Person tmp = allone[j];                    //结构变量的交换
      allone[j] = allone[j + 1];
      allone[j + 1] = tmp;
    }
  for(int k = 0; k < 6; k++)                     //输出
    cout << allone[k].name << " "
         << allone[k].id << " "
         << allone[k].salary << endl;
}//--------------------
```

运行结果为:

```
marit  27519  311
jone  12345  339
peter  23987  400
david  13916  449
yoke  12335  511
jasen  42876  623
```

程序定义了一个 Person 结构的全局数组,一共 6 个元素,每个元素都是一个 Person 结构变量。在冒泡排序中,比较成员 salary 的大小,访问相应结构成员的方法是数组元素名、点操作符加上成员名:allone[i].salary。

排序中交换数组元素,就是交换结构变量,程序中以结构变量赋值的方法来进行交换。



结构变量相互交换，在结构很大(成员很多)时，并不是一种好办法。可以建立一个结构指针数组来实现同样的功能。即，一个数组中，每个元素都是一个结构指针。

例如，下面的程序建立了一个结构指针数组，实现 ch10_5.cpp 的结构排序：

```
//---------------------
//    ch10_6.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //-------------------
Person allone[6] = {{"jone", 12345, 339.0},
                   {"david", 13916, 449.0},
                   {"marit", 27519, 311.0},
                   {"jasen", 42876, 623.0},
                   {"peter", 23987, 400.0},
                   {"yoke", 12335, 511.0}};
//---------------------
int main(){
  Person* pA[6] = {&allone[0], &allone[1], &allone[2],
                  &allone[3], &allone[4], &allone[5]};
  for(int i = 1; i<6; i++)
  for(int j = 0; j<=5-i; j++)
    if(pA[j]->salary>pA[j+1]->salary)     //比较工资成员
    {
      Person tmp = pA[j];
      pA[j] = pA[j+1];
      pA[j+1] = tmp;
    }
  for(int k = 0; k<6; k++)
    cout<<pA[k]->name<<" "
        <<pA[k]->id<<" "
        <<pA[k]->salary<<endl;
}//--------------------
```

运行结果为：

```
marit  27519  311
jone  12345  339
peter  23987  400
david  13916  449
yoke  12335  511
jasen  42876  623
```

从程序中看到，建立了一个结构指针数组 pA 并依次初始化为结构数组元素的地址值。对结构成员的访问由点操作符改成了箭头操作符，因为现在是对结构指针进行操作。

发生交换时，并不是两个结构变量值交换，而是两个结构指针交换，所以交换的临时变量是一个结构指针。最后输出的是通过结构指针访问的结构变量值。运行结果与程序

ch10_5.cpp 相同。

由于不必用结构值赋值的方法来交换,运行效率较 ch10_5.cpp 程序高。

10.4 传递结构参数

1. 传递结构值

结构可以按值传递,这种情况下整个结构值都将被复制到形参中去。

例如,下面的程序说明怎样将结构值作为参数传给函数:

```
//---------------------
//    ch10_7.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
void Print(Person pr){
  cout << pr.name << " "
      << pr.id << " "
      << pr.salary << endl;
}//--------------------
Person allone[4] = {{"jone", 12345, 339.0},
                   {"david", 13916, 449.0},
                   {"marit", 27519, 311.0},
                   {"yoke", 12335, 511.0}};
//---------------------
int main(){
  for(int i = 0; i < 4; i++)
    Print(allone[i]);
}//--------------------
```

运行结果为:

```
jone   12345   339
david   13916   449
marit   27519   311
yoke   12335   511
```

Print()函数的参数是 Person 结构变量,main()函数调用了 4 次 Print()函数,实参为 Person 结构数组的元素。

2. 传递结构的引用

结构也可以引用传递,这种情况下仅仅把结构地址传递给形参。引用传递效率较高,因为它不用传递整个结构变量的值(有时候是很大的空间),节省了传递的时间和存储空间,同时又不影响其功能。

例如，下面的程序是修改了程序 ch10_7.cpp，以引用传递结构：

```
//----------------------
//     ch10_8.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
void Print(Person& pr){
  cout << pr.name << " "
       << pr.id << " "
       << pr.salary << endl;
}//--------------------
Person allone[4] = {{"jone", 12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
                    {"yoke", 12335, 511.0}};
//----------------------
int main(){
  for(int i = 0; i < 4; i++)
    Print(allone[i]);
}//--------------------
```

运行结果为：

```
jone  12345  339
david  13916  449
marit  27519  311
yoke  12335  511
```

程序中引用传递与值传递的差别，只在 Print()函数声明的参数中结构类型名后加上一个“&”引用声明符，而函数的实现与函数调用代码都与值传递相同，所以引用传递在程序的理解上与值传递一样容易。虽然结构值还可以通过结构指针传递，但程序的可读性比引用传递要差一些(见 9.4 节)。

由于结构的大小是随用户定义而定的，所以有时候会很大。当结构很大时，引用传递的优越性才真正开始体现。引用传递的进一步介绍在 18.3 节。在编程经验上，除非是小结构，一般很少按值传递。

10.5 返回结构

1. 返回结构值

一个函数可以返回一个结构值，即若干数据类型值的一个聚集。这使得我们在 10.1 节中讨论的职工数据返回问题能得以解决。

例如，下面的程序通过函数返回一个结构值给结构变量赋值：




```
//----------------------
//    ch10_9.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
Person GetPerson(){
  Person temp;
  cout<<"Please enter a name for one person:\n";
  cin>>temp.name;
  cout<<"Please enter one's id number and his salary:\n";
  cin>>temp.id>>temp.salary;
  return temp;
}//---------------------
void Print(Person& p){
  cout<<p.name<<" "
      <<p.id<<" "
      <<p.salary<<endl;
}//---------------------
int main(){
  Person employee[3];
  for(int i=0; i<3; i++){
    employee[i]=GetPerson();
    Print(employee[i]);
  }
}//---------------------
```

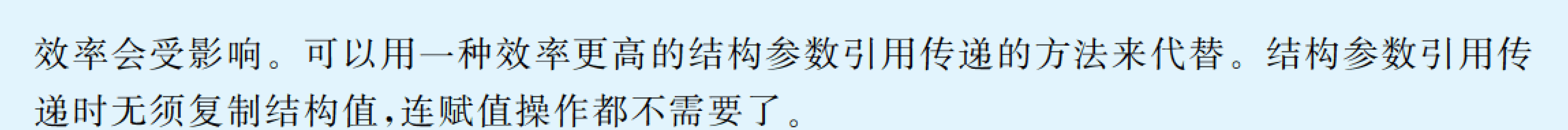
运行结果为：

```
Please enter a name for one person:
marit
Please enter one's id number and his salary:
27519  311.0
marit  27519  311
Please enter a name for one person:
jone
Please enter one's id number and his salary:
12345  339.0
jone  12345  339
Please enter a name for one person:
peter
Please enter one's id number and his salary:
23987  400.0
peter  23987  400
```

主函数中调用了 GetPerson()函数，该函数返回 Person 结构值，该结构值被赋给了主函数中的结构数组元素。

2. 结构的引用参数返回

由于结构返回时，要复制结构值给一个临时结构变量(见 9.6 节)，当结构很大时，运行



效率会受影响。可以用一种效率更高的结构参数引用传递的方法来代替。结构参数引用传递时无须复制结构值,连赋值操作都不需要了。

例如,下面的程序用结构参数引用传递的方法实现程序 ch10_9.cpp 的同样功能:

```
//----------------------
//    ch10_10.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
struct Person{
  char name[20];
  unsigned long id;
  float salary;
}; //--------------------
void GetPerson(Person& p){        //结构参数引用传递的函数
  cout <<"Please enter a name for one person:\n";
  cin >> p.name;
  cout <<"Please enter one's id number and his salary:\n";
  cin >> p.id >> p.salary;
}//---------------------
void Print(Person& p){
  cout << p.name <<" "
      << p.id <<" "
      << p.salary << endl;
}//---------------------
int main(){
  Person employee[3];
  for(int i = 0; i < 3; i++){
    GetPerson(employee[i]);      //调用后,employee[i]被赋值
    Print(employee[i]);
  }
}//---------------------
```

运行结果为:

```
Please enter a name for one person:
marit
Please enter one's id number and his salary:
27519   311.0
marit   27519   311
Please enter a name for one person:
jone
Please enter one's id number and his salary:
12345   339.0
jone   12345   339
Please enter a name for one person:
peter
Please enter one's id number and his salary:
23987   400.0
peter   23987 400
```

用结构参数引用传递,无须返回结构值,无须给另一个结构变量赋值,也无须在函数返回时复制结构值给临时结构变量,节省了系统开销。在主函数中,调用该函数的格式应作相应的调整。



3. 返回结构的引用

一个函数可以返回一个结构的引用和结构的指针(见 9.5 节),但是不要返回一个局部结构变量的引用或指针。

例如,下面的代码不应将局部结构变量的引用返回给上层函数:

```
Person& GetPerson()
{
   Person p;
   cout <<"Please enter a name for one person:\n";
   cin.get(p.name);

   cout <<"Please enter one's id number and his salary:\n";
   cin >> p.id >> p.salary;
   return p;     //局部结构变量的地址
}

int main()
{
   Person& sp = GetPerson();
   //...
}
```

在主函数中,引用 sp 用返回引用的 GetPerson()函数来初始化,使得 sp 与 GetPerson()中的局部变量名同享一个空间(见 9.6 节),这不是好的程序设计。

10.6 链表结构

1. 结构的嵌套

结构可以嵌套,即结构中可以包含结构成员。例如:

```
struct Education
{
   char major[20];
   char degree[20];
   double gpa;
};

struct Student
{
   Education school;     //结构中嵌套一个 Education 结构
   char id[20];
   int graduate;
};

Student ss;              //创建一个结构变量
```

在引用嵌套结构的成员时,要使用多个点操作符,例如 ss.school.major、ss.school.degree 等。

结构不可以递归嵌套,即**结构成员不能是自身的结构变量,但可以用自身结构指针作为成员**。因为结构自身尚在类型描述中,不能用尚无成型的结构名来描述其成员。但结构指


针的实体大小是确定的，故可以作为成员。

例如，下面的结构含有一个自身结构的指针，但不应有自身结构的变量：

```
struct List
{
   char name[20];
   List * pN;      //List 结构指针作为其成员
   List m;         //error 不应含有自身结构变量
};
```

2. 遍历结构变量的问题

我们可以用循环语句遍历结构数组中的所有元素来查找所要的结构变量。

在程序中，经常要用到多个相同结构类型的元素，而元素的个数要到运行时才能确定。同样我们能依赖动态内存分配来定义结构数组。这样，所有的结构变量在内存中依次排列，我们仍能很方便地遍历所有结构变量，查找所要的结构值。

然而，在很多情况下，我们自始至终不知道究竟需要多少个结构记录。例如，商场里客户购物，来一个就建立（动态分配）一个客户记录。临下班时，要汇总（遍历所有记录）营业额。当然我们用现成的数据库系统的软件可以很好地做这项工作，但数据库系统的内部实现正是在动态分配技术基础上实现的。

这时候我们在程序中所面临的结构记录内存布局是随机的，结构与结构之间没有联系，见图 10-1。

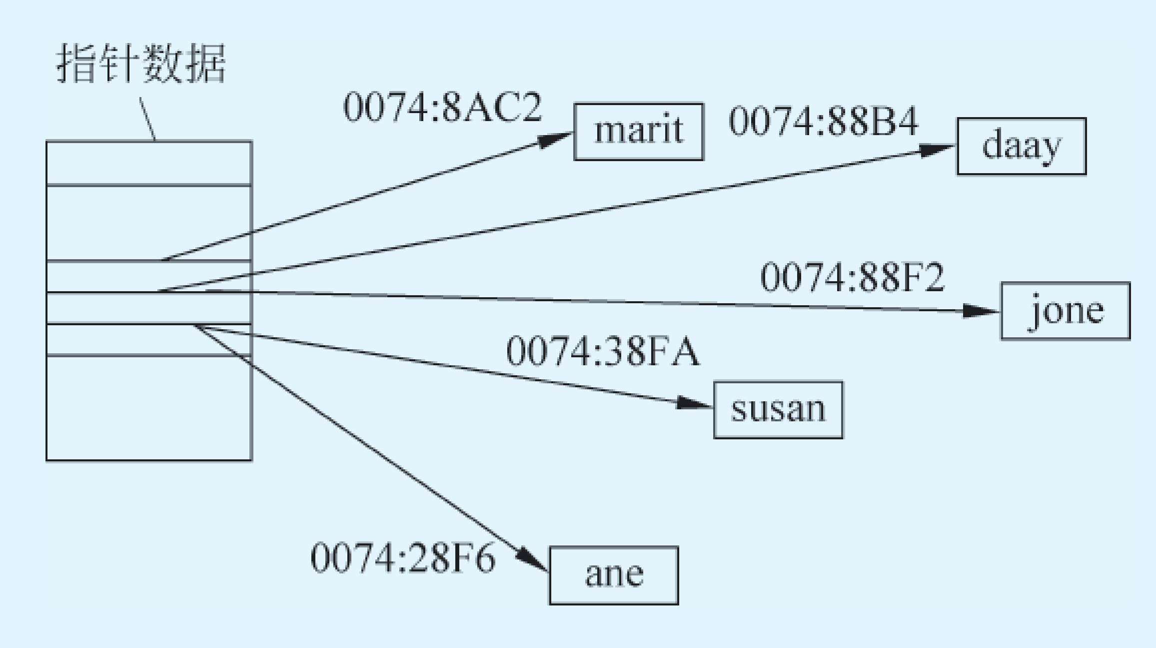


图 10-1　结构变量在内存中的布局

3. 链表结构

通过含有一个自身结构的指针，我们可以实现随机分布的结构变量的遍历。

对于下面的结构：

```
struct List
{
   char name[20];
   List * pN;
};
```

name 成员含有结构中的实际信息，pN 成员是指向另一个 List 的指针。这种结点（结构的实例）通过每个 List 的 pN 成员链接起来，能用于构造任意长的结构链，这样的结构链

称为链表。链表中的每个 List 结构变量称为结点。链表中的第一个 List 结点经常由一个指向 List 的指针引导。这个结构指针(称为链首指针)不是自身结构成员,但它指向第一个结点,而该结点的 pN 成员又指向另一个结点,而另一个结点的 pN 成员又指向一个结点,这样一直指下去,直到最后一个结点。最后一个结点的 pN 成员值为空(NULL),见图 10-2。

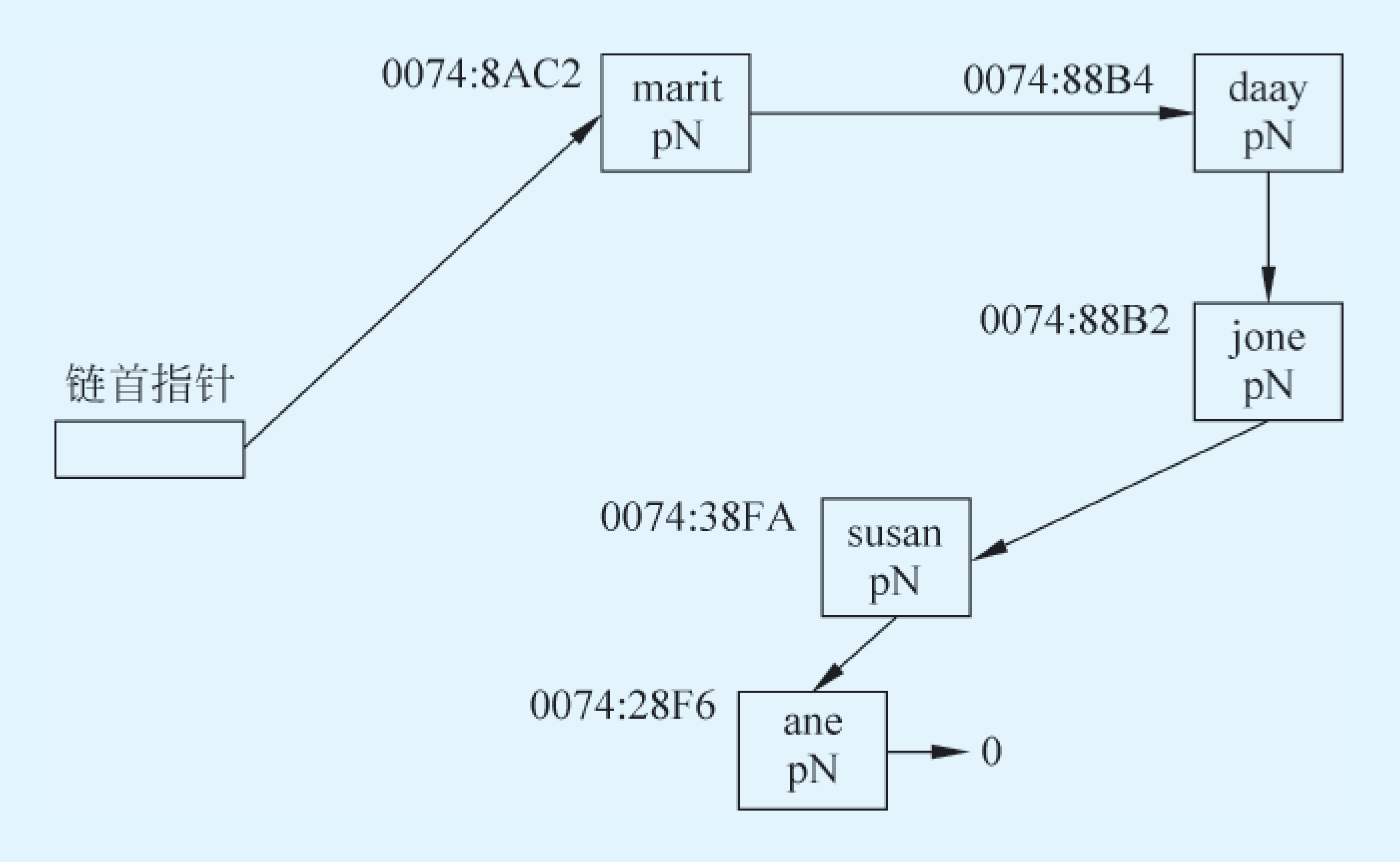


图 10-2　链表结构的描述

链表的组成是一个个链接的结点,每个结点是同类型的结构变量。可以通过程序的办法来建立和显示链表,可以插入、删除及增加结点来维护一个链表。

一个链表总是包含一个链首指针。操作链表时,一般先由链首指针引导。

10.7　创建与遍历链表

例如,下面的例子建立一个链表,并输出链表:

```
//----------------------
//     ch10_11.cpp
//----------------------
#include<iostream>
#include<iomanip>
using namespace std;
//----------------------
struct Student{
  long number;
  float score;
  Student* next;
}; //--------------------
Student* head;                         //链首指针
//----------------------
Student* getNode(){
  int num;
  float sc;
  cin>>num>>sc;
  if(num==0)
    return NULL;                       //结点无效,结束创建过程
```

```
    Student * p = new Student;
    p->number = num;
    p->score = sc;
    p->next = 0;
    return p;
}//--------------------
void Create(){
    if((head = getNode()) == 0)                    //新建第一个结点,挂入链首
      return;                                      // 返回空链表
    for(Student * pE = head, * pS; pS = getNode(); pE = pS) //pE 指向尾结点
      pE->next = pS;
}//--------------------
void ShowList(){
    cout <<"now the items of list are \n";
    for(Student * p = head; p; p = p->next)
      cout << p->number <<","<< p->score << endl;
}//--------------------
int main(){
    cout << fixed << setprecision(1);
    Create();
    ShowList();
}//--------------------
```

运行结果为：

```
54  3.4
23  3.2
24  3.5
15  4.1
66  4.0
0   0.0
now the items of list are
54,3.4
23,3.2
24,3.5
15,4.1
66,4.0
```

在主函数中，先调用 Create 函数，从无到有创建一个链表。然后调用 ShowList 函数，输出整个链表。全局指针 head 是链首指针，它在各种链表操作(插入、删除、输出)中，作为关键而又公共的数据被大家取用。

在 Create()函数中，首先调用 getNode 函数，新建一个结点给 head 指针。函数 getNode 如果返回 NULL，表示后面再无新结点。所以 Head 自然只能获得 NULL 值。如果 getNode 函数返回从堆中申请到的结点地址，则说明已新建结点，循环插入链尾，直到再无新结点。Create 函数结束后，head 指针则指向一个由堆中申请的各个结点链接而成的链首。表 10-1 是 Create()函数创建链表的过程。

ShowList()是输出链表函数，它取得链首指针 head，输出链表每个结点。它需要定义另一个 p 指针，从 head 开始，遍历链表结点。显然，不能用 head 指针去遍历结点，以防止链首地址丢失。



这里分配一个结点的 new 后面只跟一个数据类型,比 malloc()函数要简捷。关于 new,后面 14.2 节和 14.3 节还将继续深入介绍。

表 10-1　Create()函数中链表创建的过程

进入循环之前	head → NULL pEnd ps → 54 | 3.4 →
第一次进入循环,到达 s 点	head → 54 | 3.4 → pEnd ps
第一次循环结束	head → 54 | 3.4 → pEnd ps → 23 | 3.2 →
第二次循环到达 s 点	head → 54 | 3.4 → 23 | 3.2 → pEnd ps
第二次循环结束	head → 54 | 3.4 → 23 | 3.3 → pEnd ps → 24 | 3.5 →
第三次循环到达 s 点	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → pEnd ps
第三次循环结束	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → pEnd ps → 15 | 4.1 →
第四次循环到达 s 点	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → 15 | 4.1 → pEnd ps
第四次循环结束	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → 15 | 4.1 → pEnd ps → 66 | 4.0 →
第五次循环到达 s 点	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → 15 | 4.1 → 66 | 4.0 → pEnd ps
第五次循环结束	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → 15 | 4.1 → 66 | 4.0 → pEnd ps → 0 | 0.0 →
退出 Create()函数时	head → 54 | 3.4 → 23 | 3.2 → 24 | 3.5 → 15 | 4.1 → 66 | 4.0 → NULL



10.8 删除链表结点

链表结点的删除操作要保证不破坏链表的链接关系。一个结点删除后，它的前一结点的指针成员应指向它的后一结点，这样就不会因删除而使链接中断。

删除往往还包含查找子过程，因为首先要找到想要删除的结点。它包含找不到的处理。

例如，下面的代码是删除结点的函数，它删除学号为 number 的结点：

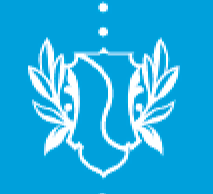
```
void Delete(long num)
{
  if(!head){                              //链空
    cout<<"\nList null\n";
    return;
  }
  Student* pGuard=head;
  if(pGuard->number==num){                //要删除的结点在链首
    head=head->next;                      //修改全局链首指针
    delete pGuard;
    cout<<num<<" the head of list have been deleted\n";
    return;
  }
  for(Student* p=pGuard->next; p; pGuard=p,p=p->next){
    if(p->number==num){                   //p指向的结点要删除吗
      pGuard->next=p->next;               //前后结点相链,跨过待删结点
      delete p;
      cout<<num<<" have been deleted\n";
      return;
    }
  }
  cout<<num<<" is not found!\n";          //到达此处表示未找到想删的结点
}
```

对于程序 ch10_11.cpp，如果主函数改成：

```
int main()
{
  Create();
  ShowList();
  Delete(54);
  ShowList();
}
```

将得到下面的输出：

```
54 3.4
23 3.2
24 3.5
15 4.1
66 4.0
0 0.0
now the items of list are
54,3.4
23,3.2
```




```
24,3.5
15,4.1
66,4.0
54 the head of list have been deleted
now the items of list are
23,3.2
24,3.5
15,4.1
66,4.0
```

相应的操作见图 10-3。

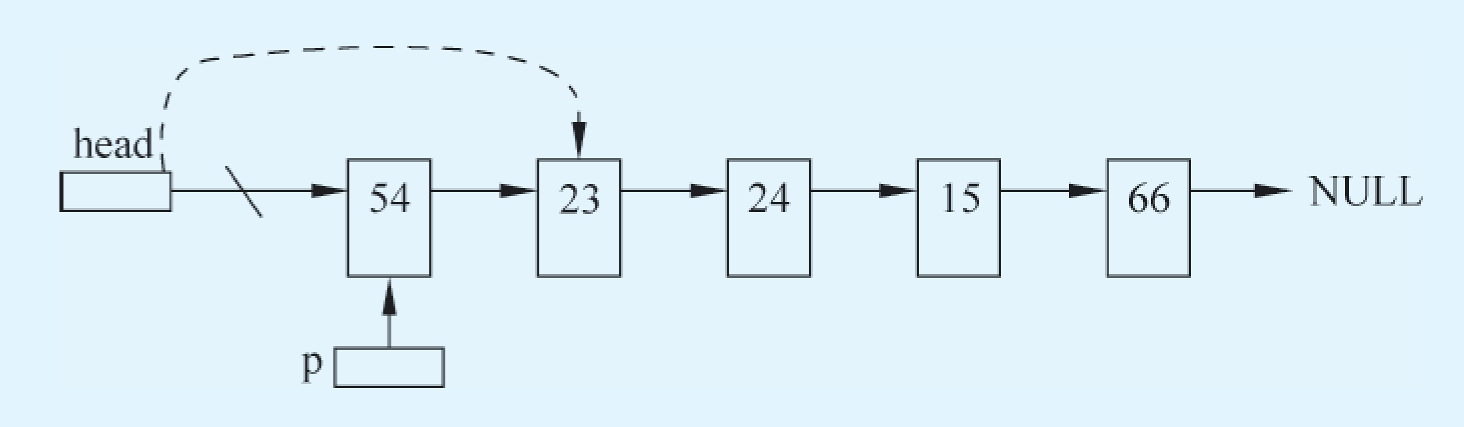


图 10-3　删除链首结点

删除链首结点的步骤为：

(1) p 指向链首结点；

(2) head 指向链首结点的下一个结点；

(3) 删除 p 指向的结点。

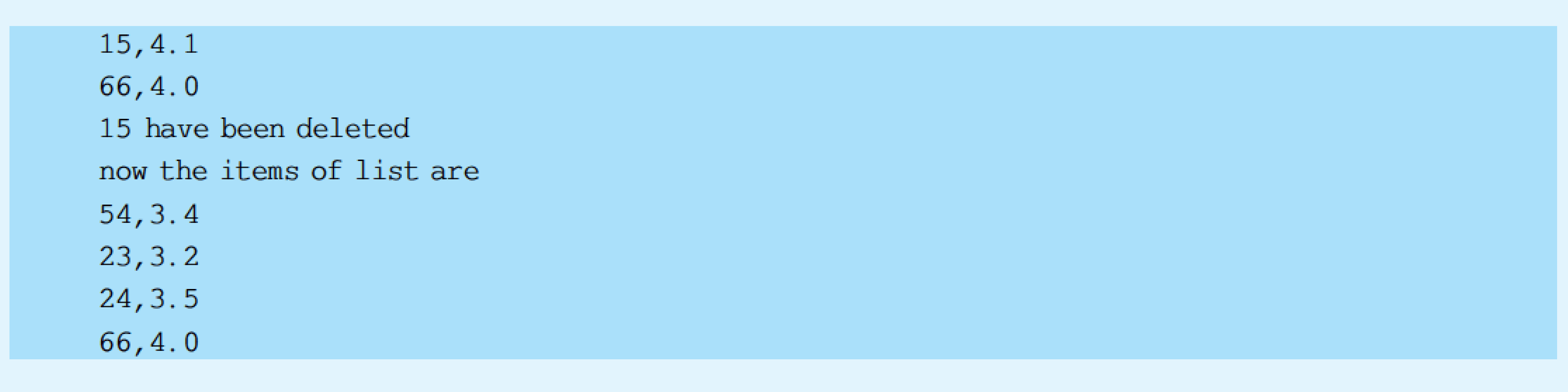
这里顺序是重要的。如果 p 不指向链首结点，则当 head 指向链首结点的下一个结点后，原链首结点脱链，导致结点地址丢失，以致无法释放堆空间。如果先删除链首结点，则 head 指针无法指向由链首结点导出的下一个结点地址。

Delete()函数中的循环语句处理非链首结点的删除。对于程序 ch10_11.cpp，如果主函数改成：

```
int main()
{
  Create();
  ShowList();
  Delete(15);
  ShowList();
}
```

将得到下面的输出：

```
54 3.4
23 3.2
24 3.5
15 4.1
66 4.0
0 0.0
now the items of list are
54,3.4
23,3.2
24,3.5
```



```
15,4.1
66,4.0
15 have been deleted
now the items of list are
54,3.4
23,3.2
24,3.5
66,4.0
```

相应的操作见图10-4。

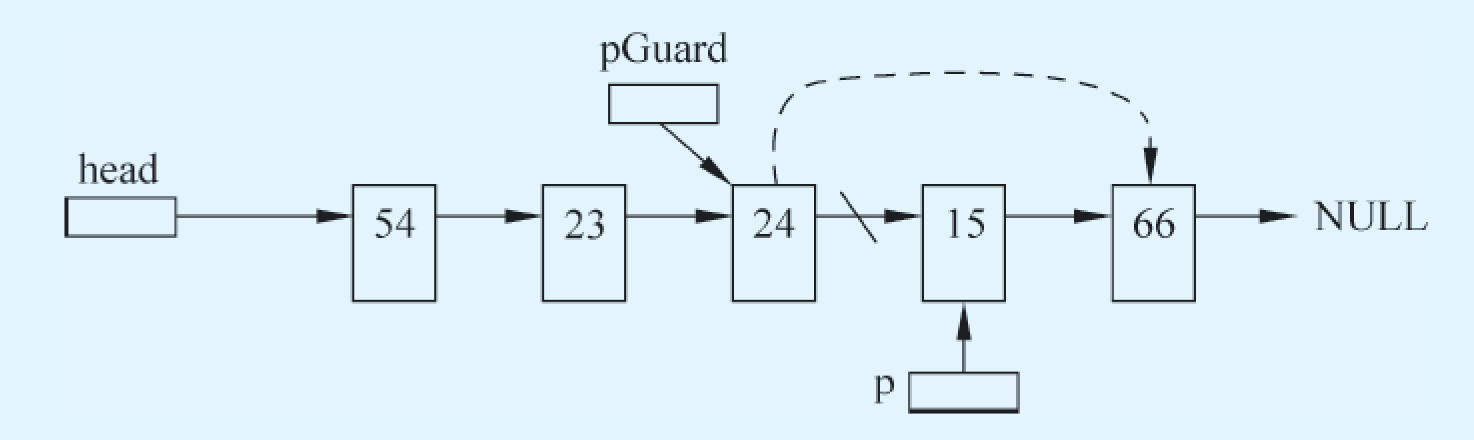


图10-4　删除非链首结点

如果主函数中对Delete()的调用改成Delete(head,11),将会得到下面的输出:

```
54 3.4
23 3.2
24 3.5
15 4.1
66 4.0
0 0.0
now the items of list are
54,3.4
23,3.2
24,3.5
15,4.1
66,4.0
11 not found!
now the items of list are
54,3.4
23,3.2
24,3.5
15,4.1
66,4.0
```

当Delete()函数找到要删除的结点时,删除的步骤为:

(1) p指向待删的结点;

(2) pGuard所指向结点的next成员指向待删结点的下一个结点;

(3) 删除p指向的结点。

在找到待删结点时,pGuard指向待删结点的前一个结点是重要的。否则,待删结点的前一结点地址丢失,其next成员无法与待删结点的后一结点链接。在数据结构课程中,通常称pGuard指针为“哨兵”。

10.9 插入链表结点

插入操作不能破坏链接关系。应将插入结点的 next 成员指向它的后一个结点,然后将前一结点的 next 成员指向插入的结点,这样就得到了新的链表。

插入操作也包含一个插入位置查找的子过程,同样也面临链表是空表或插入到链首的特殊情况。

假设创建(调用 Create())链表时,结点是按 number 值由小到大顺序排列,则插入结点函数 Insert()设计如下:

```
void Insert(Student& stud)
{
  Student * pS = new Student;                    //链结点统一由堆空间结点构成
  * pS = stud;                                   //复制成堆结点
  if(!head || pS->number < head->number){        //链表为空或插在链首
    pS->next = head;
    head = pS;
    return;
  }
  Student * pGuard = head;
  for(Student * p = pGuard->next; p; pGuard = p,p = p->next) //欲插在 pGuard 与 p 中间
    if(pS->number < p->number){
      pS->next = p;
      pGuard->next = pS;
      return;
    }
  pGuard->next = pS;                             //直到链尾还未插入,则插在最后
  pS->next = NULL;
}
```

对于程序 ch10_11. cpp,如果主函数改成:

```
int main()
{
  Create();
  Student ps = {36,3.8};
  Insert(ps);
  ShowList();
}
```

将得到下面的输出:

```
15  4.1
23  3.2
24  3.5
54  3.4
66  4.0
0   0.0
now the items of list are
15,4.1
23,3.2
24,3.5
```

```
36,3.8
54,3.4
66,4.0
```

相应的操作见图 10-5。

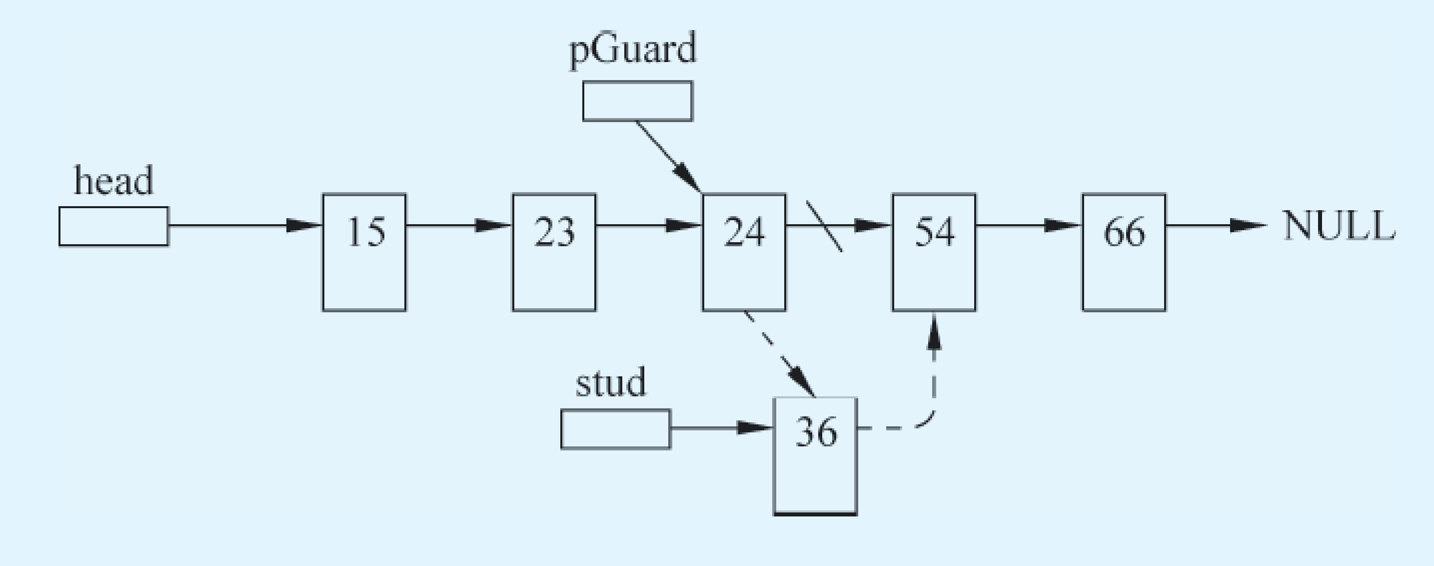


图 10-5　插入 stud 所指向的结点

链表结点统一由堆空间申请而获得，统一的好处是善后方便，统一返还，而且不会因为数据区域不同，在跨越函数时，弄出运行错误。函数 Insert()的参数来自主函数实参的引用，数据属于栈空间，不能作为链表的结点。所以 Insert 函数一上来就申请堆结点，复制参数中的数据，据此便有结点指针 pS。

如果链表为空，则只要简单地将 pS 结点指空（链尾），而后自己被链首指针指向。

如果插入的结点恰在链首，则只要将 pS 结点指向 head 指向处，其自身被链首指向。上面两种操作其实是一样的，所以放在一起实现：

```
pS->next = head;
head = pS;
```

当要插入在链表中间时，首先要找到插入位置。当发现前结点（pGuard 所指结点）的下一个结点（p 所指结点）值比插入结点值大时，就找到了插入位置。也即在 pGuard 结点与 p 结点之间插入：

（1）将 pS 结点指向 p 结点：pS->next = p；

（2）将 pGuard 结点指向 pS 结点：pGuard->next = pS；

10.10　结构应用：Josephus 问题

小孩围成圈，用环链表来表达是最自然不过的。每个结点代表一个小孩。每个结点是一种结构，内含小孩属性（编号）和结构指针（指向下一个小孩）。

一旦构成了环链，数小孩的事就是在环链中依次经历结点。小孩的离开就是代表该小孩的结点从链中删除。删除后的链表仍是环链表。

不断数小孩到一定个数时，便删除，如此循环，直到只剩最后一个。该结点自己指向自己。此时，该结点代表的小孩就是胜利者。

为了进行链表的删除，需要有“哨兵”保护结点，该结点指向当前被删结点，见图 10-6。

由于是环链表，所以没有链首指针，只有当前指针 pCurrent。但作为链表主体的数组，其第一个元素的地址必须要记住，因为从堆中分配的空间有义务要归还（用 pJose 指向数组



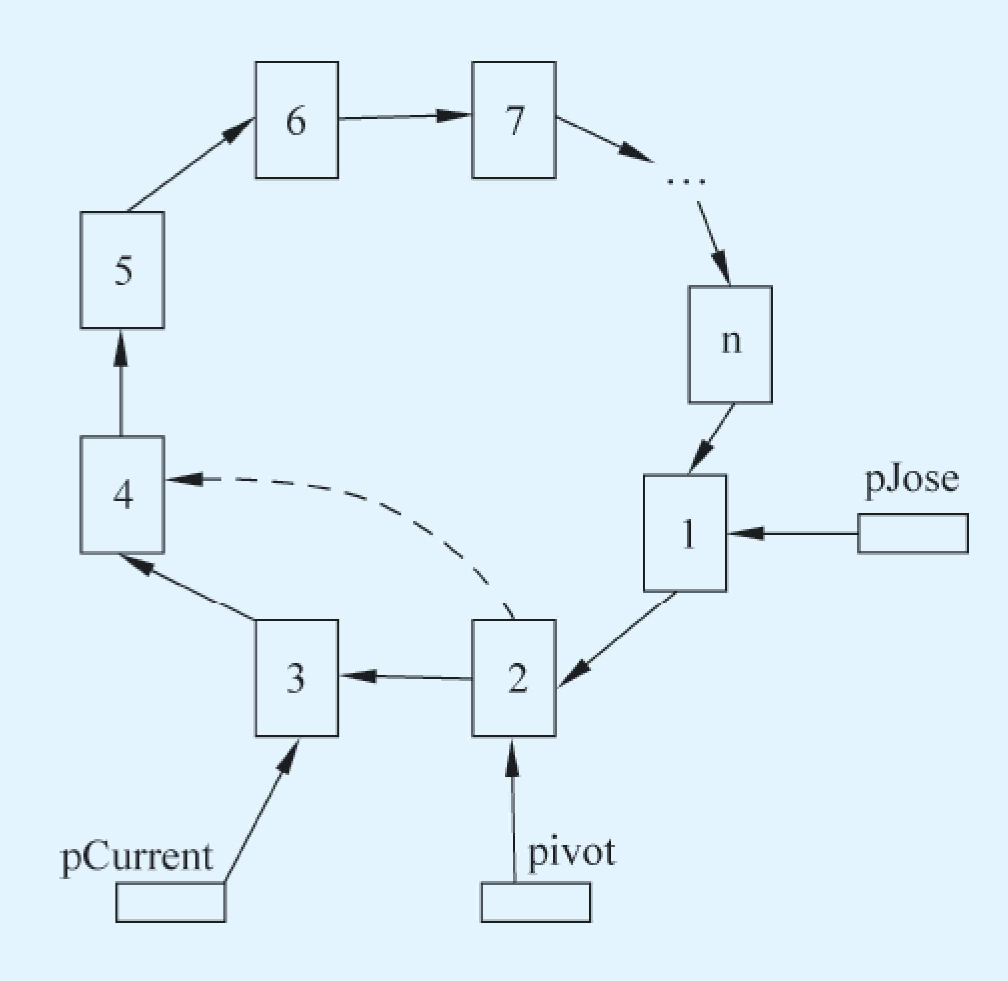


图 10-6　数完第一次小孩数，准备删除结点的情形

首地址)。

删除当前结点(pCurrent 指向的结点)意味着哨兵指向的结点要“跨过”当前结点(图 10-6 中的虚线)，然后当前结点指针 pCurrent 立即指向哨兵结点。下一次数小孩则循着虚线开始。试看 Josephus 问题解法第二版：

```
//-----------------------
//    Josephus 问题解法二
//    jose2.cpp
//-----------------------
#include<iostream>
using namespace std;
//-----------------------
struct Jose{                                        //小孩结点
  int code;
  Jose* next;
}; //---------------------
int main(){
  cout<<"please input the number of boys,\n"        //小孩数

      <<" interval of counting:\n";                 //数小孩个数
  int nBoys, interval;
  cin>>nBoys>>interval;                             //赋初值

  Jose* pJose = new Jose[nBoys];                    //从堆内存获取小孩结构数组
  for(int i = 0; i<nBoys; i++){                     //初始化结构数组：环链，编号，输出
    cout<<(i%10 ? " ":"\n ")<<i+1;                  //元素按 10 个一行输出
    pJose[i].code = i+1;                            //小孩编号
    pJose[i].next = &pJose[(i+1)%nBoys];            //链到下一个元素
  }
  Jose* pivot;                                      //链表哨兵
  Jose* pCurrent = &pJose[nBoys-1];                 //下一结点就是第一个小孩
  for(int i = 0; i<nBoys-1; ){                      //处理未获胜的所有小孩
    for(int j = interval; j--; pCurrent = pCurrent->next) //数小孩
      pivot = pCurrent;
    cout<<(i++%10 ? " " : "\n ")<<pCurrent->code; //输出离队小孩
```

```
      pivot->next=pCurrent->next;                    //小孩脱链
    }
    cout<<"\n\nthe winner is "<<pivot->code<<endl;   //获胜者
    delete[] pJose;                                  //返还堆空间
}//--------------------
```

运行结果为：

```
please input the number of boys,
          interval of counting:
15  3
     1   2   3   4   5   6   7   8   9   10
          11   12   13   14   15
     3   6   9   12   15   4   8   13   2   10
     1   11   7   14

the winner is 5
```

程序从堆中分配小孩结构数组空间，因而可以在运行时确定小孩数，给求解问题带来了灵活性。

在初始化链表的时候，将小孩数组顺序地链接起来，直至最后一个结点链到第一个结点，构成环链表。在环链表的形成过程中，给小孩编号和依次输出全体小孩。

如果起点可以是任意一个小孩，则程序应作如何修改呢？

小结

结构是一种用户定义的数据类型，声明结构时，不产生内存的分配，只有在定义结构变量时，才分配内存空间。

结构作为参数传递时，其值进行了复制，当结构很大时，宜采用结构的引用传递。这时候，结构仅仅传递地址，既省时又省空间。

结构的概念是理解各种数据结构的关键。我们在这里处理的链表称为单向链表，因为这个链表只能从一端向另一端遍历整个链表，反之则不行。此外，还有双向链表、队列、栈、树等，这些内容在数据结构课程中进一步研究。

在堆中分配结构空间时，我们看到 new 操作符比 malloc()函数的使用更简便。

在 C++中，结构是类的过渡，类的功能涵盖了结构的一切。当我们在讨论结构的时候，我们往往用结构变量这个词；在讨论类的时候，我们用对象这个词。因为结构中仅仅包含各种数据变量，而类中不但包含数据变量，还包含对数据变量的操作。在以后各章将对类进行充分的描述。

在 C++中，结构在不断退化，因为类可以代表自定义数据类型的一切。与其相关的联合(union)和位段操作也在退化，它们更多地用在与硬件控制有关的低级程序设计中。

用环链表结构来解决 Josephus 问题是比较适宜的，因为删除结点模拟小孩脱链比较形象，程序比较容易描述和理解。但是程序设计中，将所有算法细节都放在一个主函数中描述，显得复杂。如果是一个大程序，应该怎样划分成若干小源程序呢？

当我们在过程化程序设计中养成了良好的编程风格，掌握了 C++语言要素，搞懂了



C++程序结构,把握了 C++函数机制,融通了指针和引用,积累了较典型的过程化程序设计经验之后,就可以轻松自在地跨入学习面向对象程序设计方法的进程。

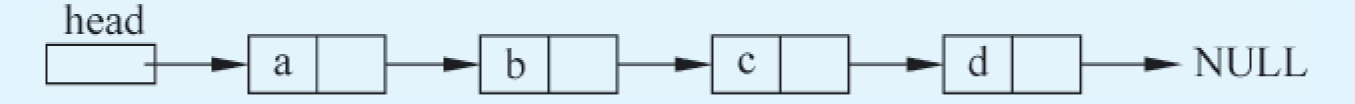
练习

10.1 利用结构类型编制程序,实现输入一个学生的数学期中和期末成绩,然后计算并输出其平均成绩。

10.2 已知 head 指向一个带头结点的单向链表,链表中每个结点包含字符型数据和指向本结构结点的指针。编写函数实现在值为“jone”的结点前插入值为“marit”的结点,若没有值为“jone”的结点,则插在链表最后。

10.3 已知 head 指向一个带头结点的单向链表,链表中每个结点包含数据 long 和指向本结构结点的指针。编写函数实现如下图所示的逆置。

原链表为:



逆置后的链表为:

head → d → c → b → a → NULL

10.4 读下面的链表操作程序。

(1) 将函数 ShowList()和 AddToEnd()改成非递归形式(可以修改函数原型)。

```
#include <iostream>
using namespace std;
struct Lnode
{
   double data;
   Lnode * next;
};

void ShowList(Lnode * list)
{
   if(list)
   {
     cout << list -> data << endl;
     if(list -> next)
        ShowList(list -> next);        //递归调用
   }
}

void AddToEnd(Lnode * new, Lnode * head)
{
   if(head == NULL)
   {
     head = new1;
     new1 -> next = NULL;
   }
```

```
    else
      AddToEnd(new1, head->next)        //递归调用
}

Lnode * GetNode()
{
    Lnode * item;
    Item = new Lnode;
    if(item)
    {
      item->next = NULL;
      item->data = 0.0;
    }
    else
      cout <<"Nothing allocated\n";
    return item;
}

int main()
{
    Lnode * head = NULL;
    Lnode * temp;
    temp = GetNode();
    while(temp)
    {
          cout <<"Data? ";
      cin >> temp->data;
      if(temp->data > 0)
         AddToEnd(temp, head);
      else
         break;
      temp = GetNode();
    }
    ShowList(head);
}
```

(2) 写出输入下面内容之后的运行结果。

```
Data? 3
Data? 5
Data? 7
Data? 6
Data? 4
Data? 8
Data? -3
```

(3) 在主函数结束之前,增加一个删除整个链表的函数调用 DeleteList(),使得从堆空间分配得到的结构变量能够返还。

10.5 定义两个同种单向链表(结点中包含一个整型数和一个指向本结点类型的指针),这两个链表中数据都已排序好,编制程序,合并这两个链表。

10.6 建立一个 10 结点的单向链表,每个结点包括学号、姓名、性别、年龄。采用插入排序法对其进行排序,按学号从小到大排列。

第二部分

面向对象程序设计



第11章　类

类构成了实现 C++面向对象程序设计的基础。类是 C++封装的基本单元，它把数据和函数封装在一起。当类的成员声明为保护时，外部不能访问；声明为公共时，则在任何地方都可以访问。学习本章后，要求掌握声明和定义类和成员函数的方法，掌握访问成员函数的方法，理解保护数据屏蔽外部访问的原理，使得对类的封装有更好的认识。

11.1　从结构到类

C 的结构可把相关联的数据元素组成一个单独的统一体。

例如，下面的代码是一个存款结构：

```
struct Savings
{
  unsigned accountNumber;    //账号
  float balance;             //结算额
};
```

Savings 结构的每个实例(对象)包含同样的两个数据元素。

例如，下面的代码定义两个结构对象：

```
void fn()
{ Savings a;                 //一个结构的实例：一个银行存款账户
  Savings b;                 //又一个结构的实例：另一个银行存款账户
  a.accountNumber = 1;       //一个银行存款的账号
  b.accountNumber = 2;       //另一个银行存款的账号
}
```

a、b 是两个不同的对象，其分量 accountNumber 对应于不同的空间，值可以不同。

任何程序，只要说明了 Savings 结构对象，就可以修改其对象中属性(分量，或数据成员)的值。

→ 在 C 中，说明结构对象的方法为：




```
struct Savings a;
```

C++中,说明方法为:

```
Savings a; //关键字 struct 不必要
```

C 的结构不含成员函数。C++的类既能包含数据成员(data member),又能包含函数成员或称成员函数(member function)。

例如,下面的代码中,Savings 类含有两个数据成员、一个成员函数:

```
class Savings
{
  public:
    unsigned deposit(unsigned amount)      //成员函数
    {
      balance += amount;
      return balance;
    }
  private:
    unsigned accountNumber;                //数据成员
    float balance;
};
```

关键字 class 表示类,Savings 是类名,一般首字符用大写字母表示,以示与对象名的区别。关键字 public 和 protected(或 private)表示访问控制。

在类中说明的,要么是数据成员,要么是成员函数。它们或者说明为 public 的,或者说明为 protected 的,或者说明为 private 的。

类具有封装性,它可以说明哪些成员是 public 的,哪些不是。说明了 protected 的成员,外部是不能访问的。

例如,下面的代码定义两个类对象,对于上面的 Savings 类定义,不能对其数据成员进行访问:

```
void fn()
{
  Savings a;            //定义类对象
  Savings b;
  a.balance = 100.5;    //error: 不能访问 balance,因为它是保护成员
  b.balance = 200.5;    //error
  a.deposit(100);       //ok: 使 balance 赋以值 100,deposit()是公共的
}
```

这里用类 Savings 来定义对象 a 和 b。由于在类 Savings 中说明了 balance 数据成员是 private(私有)的,所以在普通函数 fn()中就不可以直接访问它,而要通过访问类的 public 成员函数间接地访问。

→ 类与结构的区别

C++中,结构是用关键字 struct 声明的类,默认情况下其成员是公共(public)的。例如,Savings 类也可以如下定义:

```
struct Savings
{
  public:       //该行可省略
```

```
        unsigned deposit(unsigned amount)     //成员函数
        {
          balance += amount;
          return balance;
        }
      private:
        unsigned accountNumber;               //数据成员
        float balance;
    };
```

而 C++ 中,默认情况下类(class)定义中的成员是 private 的。

结构在 C 中不允许有成员函数,而在 C++ 中可以有成员函数。第 10 章介绍的结构是 C 中的结构,这样介绍的目的也是为了分清面向对象程序设计中的类与通常意义下的结构之本质区别。

11.2 软件方法的发展

较早的软件开发,用结构化程序设计方法。程序的定律是:

程序=(算法)+(数据结构)

定律中,算法是负责计算的主体,表示为函数或者过程;数据结构是数据描述和存储组织的结构,表示为数据类型定义和数据实体存储。该定律表示算法是一个独立的整体,数据结构也是一个独立的整体。二者分开设计,以算法(函数或过程)为主。算法与数据结构的关系见图 11-1。

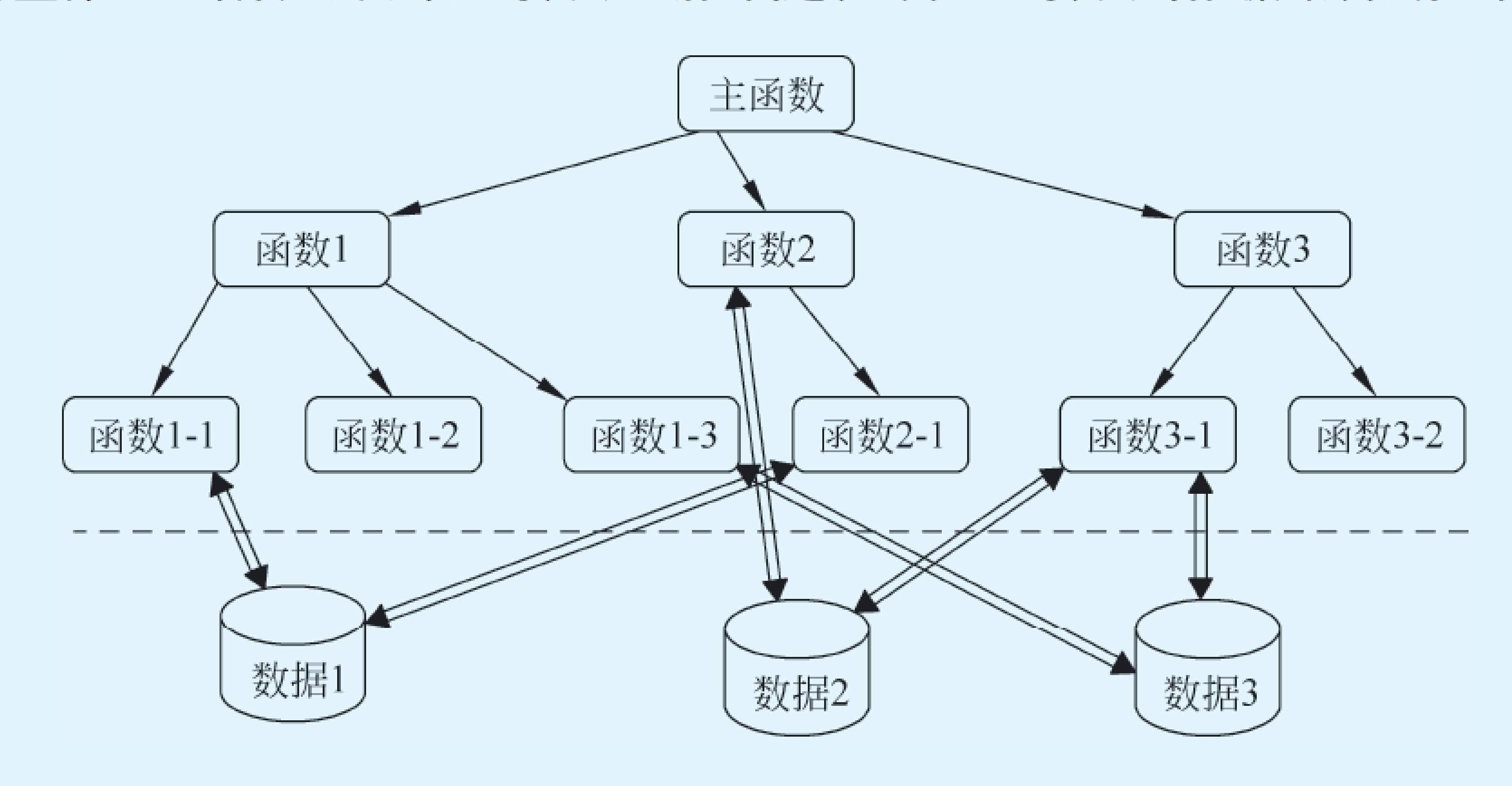


图 11-1 算法与数据结构的关系

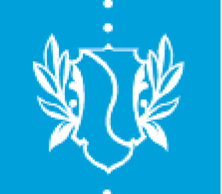
图中的虚线表示上面的算法与下面的数据分离。双箭头线表示作为黑盒的函数输入输出数据。

在前 10 章中,所有的程序都是"算法+数据结构"的典型描述。

随着时间的流逝,软件工程师越来越注重于系统整体关系的表示和数据模型技术(把数据结构与算法看作一个独立功能模块)。程序定律被重新认识:

程序=(算法+数据结构)s

即算法与数据结构是一个整体,**算法总是离不开数据结构**,算法含有对数据结构的访问,**算法只能适用于特定的数据结构**。因此设计一个算法适合于访问多个数据结构是不明智的,



而且数据结构由多个算法来对其进行同种操作也是多余的,见图 11-2 说明。

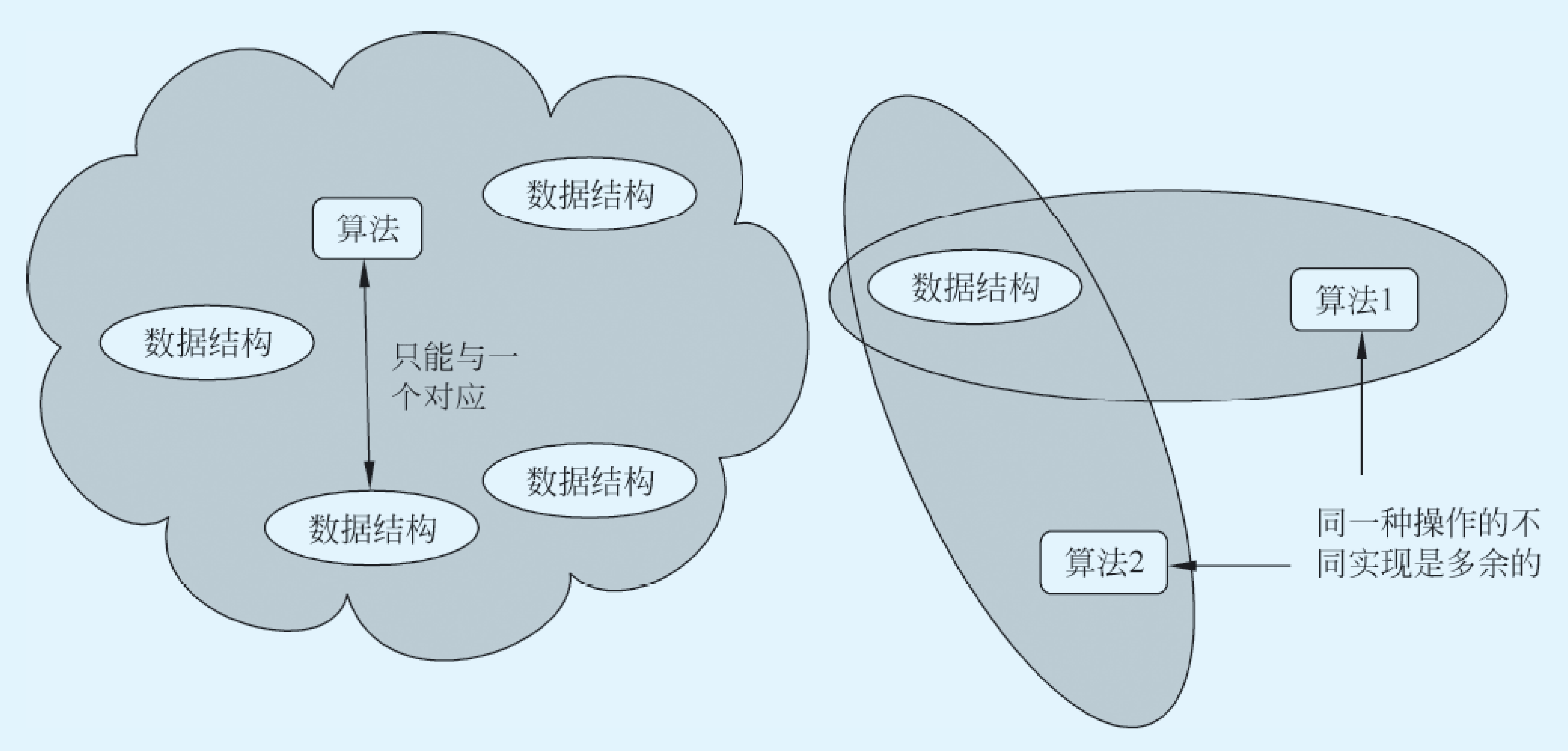


图 11-2　旧的程序定律给算法与数据结构带来不适

这是面向对象程序设计的基础,在面向对象中,算法与数据结构被捆绑成一个类,从这样的角度看问题,就不用为如何实现通盘的程序功能而费尽心机了。现实世界本身就是一个对象的世界,任何对象都具有一定的属性与操作,也就总能用数据结构与算法二者合一地来描述。这时候,程序定律被再次另眼相看:

对象＝(算法＋数据结构)

程序＝(对象＋对象＋……)

也就是说,程序就是许多对象在计算机中相继表现自己,而对象又是一个个程序实体,见图 11-3。

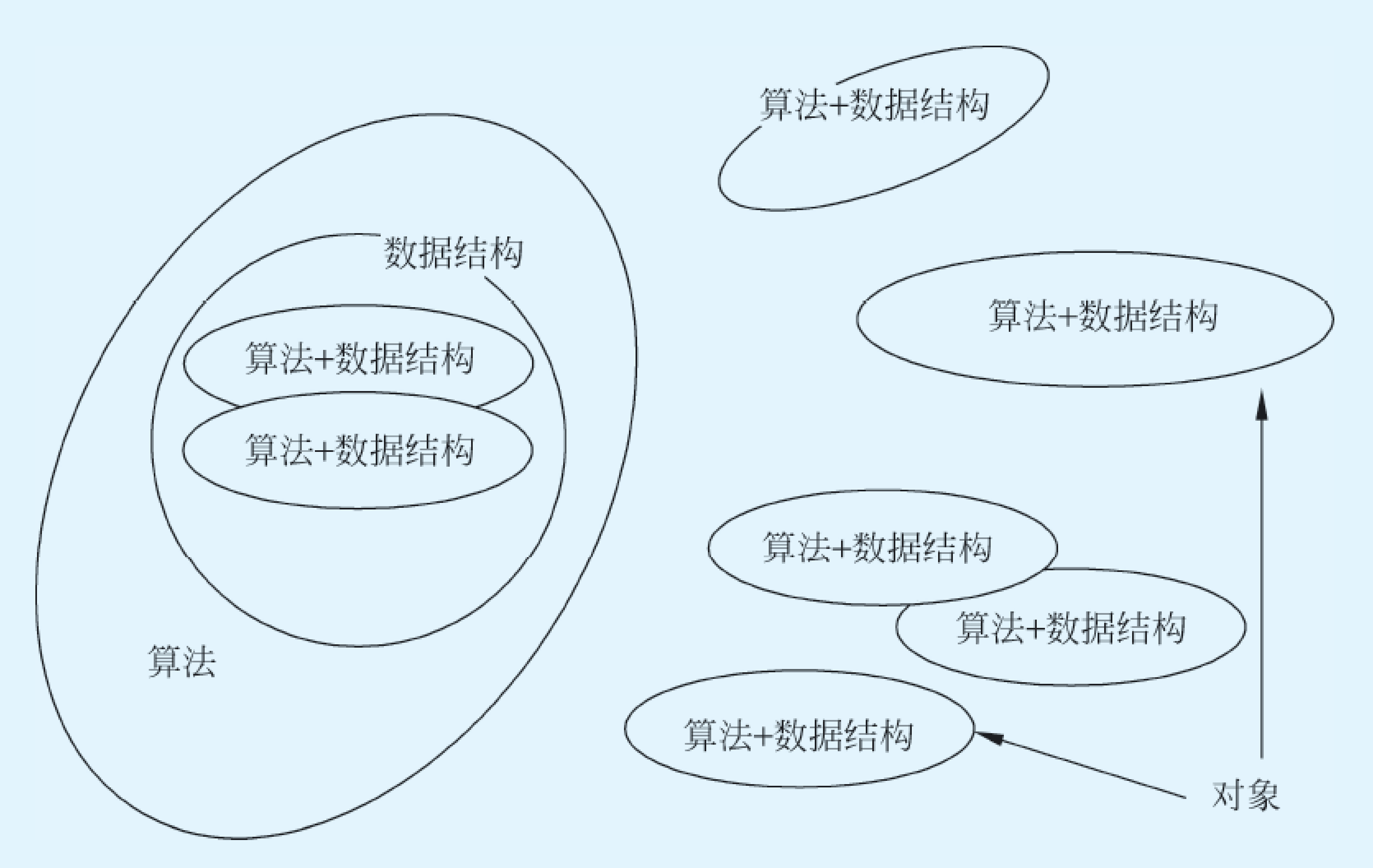
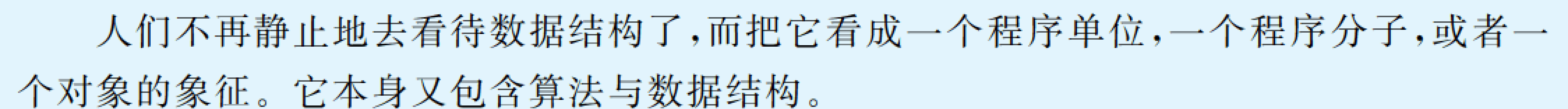


图 11-3　构成程序的对象

在此基础上,就可以依赖这些对象框架组装程序,不用煞费苦心地构建庞大的功能控制体,程序中的各个对象单元各司其职,各担其责,和谐共处。

在这期间,对象的类架构(类编程)便成为了编程中心工作。



人们不再静止地去看待数据结构了，而把它看成一个程序单位，一个程序分子，或者一个对象的象征。它本身又包含算法与数据结构。

既然对象们可以组建自己的功能结构，拼装程序而运行，那么作为对象的算法和数据结构单元便可作为新型数据结构融入程序元素。

同样，描述和组织对象数据而成为新型数据结构，就像最初组织数据，成为更高阶的程序元素。

其中，对象数据的不同细节构成对象实现的不同版本，成为了继承和多态的对象技术；不同数据集合的相同对象版本，构成了模板，成为更高阶的程序元素。

由于突破了软件设计思想的障碍，因此程序规模迅速扩大，软件产业得以飞速发展。类的实现机制就这样在程序语言中应运而生，C++是类机制实现得比较完善的一种高级语言。人们用对象的观点，抽象(见 13.1 节)一个个具有数据属性和动作的实体，直接描述于语言，进行真正意义的面向对象程序设计。

11.3 定义成员函数

1. 命名成员函数

我们从下列类的例子来展开类的讨论。下面的程序中定义了一个类，该类由 3 个公共成员函数和 3 个私有数据成员组成。在主函数中定义了一个类对象，使用了该对象(对象表现了自己)：

```
//---------------------
//    ch11_1.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Tdate{
public:
  void Set(int m,int d,int y){       //置日期值
    month = m; day = d; year = y;
  }
  int IsLeapYear(){                  //判是否闰年
    return (year % 4 == 0 && year % 100!= 0)||(year % 400 == 0);
  }
  void Print(){                      //输出日期值
    cout << month <<"/"<< day <<"/"<< year << endl;
  }
private:
  int month;
  int day;
  int year;
}; //-------------------
int main(){
  Tdate a;
  a.Set(2,4,1998);
  a.Print();
}//--------------------
```



运行结果为:

```
2/4/1998
```

主函数开始运行时,首先创建一个 Tdate 类的对象,然后调用 Tdate 的公共成员函数 Set()来给对象的数据成员赋值,再调用成员函数 Print()打印输出结果。

类定义中的 IsLeapYear()成员函数在主函数中并未用到,但其定义仍放在类中。类定义是提供给更多不同用途的程序共享的,并不受单个程序应用的影响而"优化"。这是类定义的常识。

函数 Set(int,int,int)的全名是 Tdate::Set(int,int,int)。类名 Tdate 的作用是指出 Set(int,int,int)是 Tdate 的一个成员函数,而不是其他类的成员函数(如 Teacher::Set(int)),也不是普通函数。没有类名的函数也称为非成员函数,在本章之前接触的都是非成员函数。成员函数也叫方法(mathod),它多出现于面向对象方法论的叙述中。

数据成员 month 的全名为 Tdate::month。

::叫作用域区分符,指明一个函数属于哪个类或一个数据属于哪个类。::可以不跟类名,表示全局数据或全局函数(即非成员函数)。

例如,下面的代码在成员函数 Set()中调用了非成员函数 Set():

```
int month;                      //全局变量
int day;
int year;

void Set(int m,int d,int y)     //非成员函数
{
  ::month = m;                  //给全局变量赋值,此处可省略::
  ::day = d;
  ::year = y;
}

class Tdate
{
  public:
    void Set(int m,int d,int y) //成员函数
    {
      ::Set(m,d,y);             //调用非成员函数
    }
  private:
    int month;
    int day;
    int year;
};
```

2. 在类中定义成员函数

在 ch11_1.cpp 中,类定义的大括号所包含的 3 个成员函数是在类中定义的。在类中定义的成员函数一般规模都比较小,语句只有 1～5 句,控制结构简单。它们一般为内联函数,即使没有明确用 inline 标示。

在 C++中,类定义通常在头文件中,因此这些成员函数定义也伴随着进入头文件。我们



知道函数声明一般在头文件，而函数定义不能在头文件，因为伴随着在不同程序文件中的包含展开，它们将被编译多次。如果是内联函数，包含在头文件中是允许的，因为内联函数作为全局静态属性在源程序中原地扩展。由于在类中定义的成员被默认为内联函数，所以就避免了不能被包含在头文件中的问题。

3. 在类之后定义成员函数

对于大的成员函数来说，直接把代码放在类定义中使用起来十分不便。为了避免这种情况，C++允许在其他地方定义成员函数。

例如，下面的程序将 ch11_1.cpp 中的类定义分成两部分，一部分为类定义的头文件，另一部分是类的成员函数定义，即：

```
//----------------------
//     tdate.h 类定义
//----------------------
#ifndef TDATE
#define TDATE
class Tdate{
public:
  void Set(int,int,int);                  //成员函数声明
  int IsLeapYear();
  void Print();
private:
  int month;
  int day;
  int year;
}; //--------------------
#endif

//----------------------
//     tdate.cpp 类实现 -- 成员函数定义
//----------------------
#include"tdate.h"
#include<iostream>
using namespace std;
//----------------------
void Tdate::Set(int m,int d,int y){
  month=m; day=d; year=y;
}//---------------------
int Tdate::IsLeapYear(){
  return (year%4==0&&year%100!=0)||(year%400==0);
}//---------------------
void Tdate::Print(){
  cout<<month<<"/"<<day<<"/"<<year<<endl;
}//---------------------
```

将类定义和其成员函数定义分开，是目前开发程序的通常做法。我们把类定义(头文件)看成类的外部接口，类的成员函数定义看成类的内部实现。 将类拿来编制应用程序时，只需类的外部接口(头文件)。这和我们使用标准库函数的道理是一样的，即只需包含某函数声明的头文件。因为类定义中全部包含了类中成员函数的声明。

例如,上例代码等价于下面的代码,该代码在一个文件中实现类定义和其成员函数定义:

```
#include <iostream>
using namespace std;
class Tdate
{
  public:
    void Set(int,int,int);
    int IsLeapYear();
    void Print();
  private:
    int month;
        int day;
    int year;
};

void Tdate::Set(int m,int d,int y)
{
  month = m;
  day = d;
  year = y;
}

int Tdate::IsLeapYear()
{
  return (year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}

void Tdate::Print()
{
  cout << month <<"/ " << day <<"/ " << year << endl;
}
```

在类定义的外部定义成员函数,比在类内部定义时,成员函数名前多加上一个类名。如果该函数的前面没有用"类名::"表达形式把它与该类紧紧连在一起,编译器就会认为该函数是一个普通函数,它只是与类中的成员函数有相同的名字罢了。以后在连接时,会查出缺少与成员函数相对应的定义而报错。

在类定义内部定义成员函数时,类名是省略的(如 ch11_1.cpp)。就像某人在家时,家人都叫他小宝,而出门在外时,别人都叫他范小宝。

类名加在成员函数名之前而不是加在函数的返回类型前。

例如,下面的代码不该将类名加在成员函数定义的最开头:

```
Tdate::void Set(int m,int d,int y)   //error
{
  //...
}
```

4. 重载成员函数

成员函数可以用与传统函数一样的方法重载。但由于类名是成员函数名的一部分,所


以一个类的成员函数与另一个类的成员函数即使同名，也不能认为是重载。

例如，下面的代码中定义了 Student 类的两个重载函数，定义了 Slope 类的一个成员函数，定义了一个普通函数，并在主函数中分别调用了它们：

```
 class Student
 {
   public:
     float grade(){ //... }
     float grade(float newGPA){ //...}
     //...
       protected:
         //...
     };
    class Slope
{
  public:
    float grade(){ //... }
    //...
  protected:
    //...
}t;                                    //全局对象 t
char grade(float value){ //... }

int main()
{
  Student s;
  slope t;
  s.grade(3.2);                        //Student::grade(float)
  float v = s.grade();                 //Student::grade()
  char c = grade(v);                   //::grade(float)
  float m = t.grade();                 //Slope::grade()
}
```

在主函数运行时，一共调用了 4 个 grade()函数：s.grade(3.2)匹配 Student 类中的 grade(float)；s.grade()匹配 Student 类中的 grade()；grade()函数匹配全局函数 grade(float)；t.grade()匹配 Slope 类中的 grade()。

11.4 调用成员函数

1. 调用一个成员函数

一个对象要表现其行为，就要调用它的成员函数。调用成员函数的形式类似于访问一个结构对象的分量，先指明对象，再指明分量。它必须指定对象和成员名，否则无意义。

例如，下面的代码把调用成员的形式搞错了：

```
 #include "tdate.h"
 #include <iostream>
using namespace std;
Tdate s;                         //全局对象名为 s

void func()
```




```
{
  month = 10;              //error: month 是什么,成员还是对象?
  Tdate::month = 10;       //error: month 是 Tdate 类的哪个对象?
  Tdate::Set(2,15,1998);   //error: Set 对哪个对象操作呢?
}
```

又如,下面的代码是正确的调用成员函数形式:

```
void func()                 //普通函数
{
  Tdate oneday;             //创建对象
  oneday.Set(2,15,1998);    //调用其成员函数
  oneday.Print();
}
```

2. 用指针调用成员函数

对象可以由指针来引导。例如,下面的程序用指针引出对象的成员函数。该程序是一个多文件程序结构,工程 ch11_2.prj 包含两个源文件:

```
//**********************
// ch11_2.prj
//**********************
ch11_2.cpp
tdate.cpp
//**********************
```

在 ch11_2.cpp 中,只要包含类 Tdate 的头文件 tdate.h(见 11.3 节),就可以使用该类了:

```
//----------------------
//     ch11_2.cpp
//----------------------
#include"tdate.h"
#include<iostream>
using namespace std;
//----------------------
void someFunc(Tdate* pS){
  pS->Print();          //pS 是 s 对象的指针
  if(pS->IsLeapYear())
    cout<<"oh oh\n";
  else
    cout<<"right\n";
}//--------------------
int main(){
  Tdate s;
  s.Set(2,15,1998);
  someFunc(&s);         //对象的地址传给指针
}//--------------------
```

运行结果为:

```
2/15/1998
oh oh
```

someFunc()是普通函数,所以不用在前面加对象名。s 的地址作为参数调用 someFunc(),

在 someFunc()中，s 对象通过指针尽情地表现自己。

3. 用引用传递来访问成员函数

用对象的引用来调用成员函数，看上去和使用对象自身的形式一样。

例如，下面的程序是根据 ch11_2.cpp 改编的，它使用了引用传递，而后调用成员函数。该程序同样是一个多文件程序结构，工程 ch11_3.prj 包含：

```
//**********************
// ch11_3.prj
//**********************
ch11_3.cpp
tdate.cpp
//****************** ***
```

在 ch11_3.cpp 中，也只要包含类 Tdate 的头文件 tdate.h，就可以使用该类：

```
//----------------------
//    ch11_3.cpp
//----------------------
#include"tdate.h"
#include<iostream>
using namespace std;
//----------------------
void someFunc(Tdate& refs){
  refs.Print();          //refs 是 s 对象的别名
  if(refs.IsLeapYear())
    cout<<"oh oh\n";
  else
    cout<<"right\n";
}//---------------------
int main(){
  Tdate s;
  s.Set(2,15,1998);
  someFunc(s);           //对象的地址传给引用
}//---------------------
```

运行结果为：

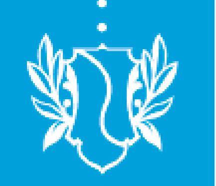
```
2/15/1998
oh oh
```

4. 在成员函数中访问成员

成员函数必须用对象来调用。另一方面，在成员函数内部，访问数据成员或成员函数无须如此。

例如，下面的程序中，成员函数内部访问了数据成员：

```
//----------------------
//    ch11_4.cpp
//----------------------
#include"tdate.h"
```




```
#include <iostream>
using namespace std;
//--------------------
void Tdate::Set(int m,int d,int y){
  month = m;      //不能在 month 前加对象名
  day = d;
  year = y;
}//--------------------
void Tdate::Print(){
  cout << month << "/" << day << "/" << year << endl; }
//--------------------
int Tdate::IsLeapYear(){
  return(year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}//--------------------
int main(){
  Tdate s;
  Tdate t;
  s.Set(2,15,1998);
      t.Set(3,15,1997);
  s.Print();
  t.Print();
}//--------------------
```

运行结果为:

```
2/15/1998
3/15/1997
```

在主函数中创建了 s 和 t 对象,然后 s 和 t 对象分别调用了成员函数 Set()。在 Set()成员函数中访问了 month、day 和 year。

一个类中所有对象调用的成员函数都是同一代码段。那么,成员函数又是怎么识别 month、day 和 year 是属于哪个对象的呢?

原来,在对象调用 s.Set(2,15,1998)时,成员函数除了接受 3 个实参外,还接受了一个对象 s 的地址。这个地址被一个隐含的形参 this 指针所获取,它等同于执行 this=&s。所有对数据成员的访问都隐含地被加上前缀 this->。所以:

```
month = m;   等价于   this->month = m;   等价于   s.month = m
```

因此,无论对应哪个对象调用,其成员函数从获得的参数就能清楚地判断是隐式的对象参数中的成员,还是显式参数名。这就是成员函数中访问成员无须对象名作前缀的原因。Set()成员函数还可表示成下列代码:

```
void Tdate::Set(int m,int d,int y)
{
  this->month = m;
  this->day = d;
  this->year = y;
}
```

但不可表示成下列代码:

```
void Tdate::Set(int m, int d, int y)
{
  s.month = m;  //error:  s 没有声明
  s.day = d;
  s.year = y;
}
```

因为成员函数是所有对象共享的代码，不是某一个对象所独占的，所以不能在成员函数内使用某个特定的对象。在编译期间，s 对象由于没有在成员函数内部或文件作用域中声明而导致失败。

→ 一个类对象所占据的内存空间由它的数据成员所占据的空间总和所决定。类的成员函数不占据对象的内存空间。

11.5 保护成员

可以把类的成员声明为保护的(protected)，这时，从类的外部(指在普通函数或其他类的成员函数中)就不能对它们进行访问；也可以把类的成员声明为公共的，这时在任何地方都可以对它们进行访问。

在类中设置保护屏障，不让外部访问，主要是由面向对象程序的目标决定的：

(1) 相对于外部函数(普通函数，其他类的成员函数)而言，保护类的内部数据不被肆意侵犯。电视机通过一个接口把它提供给外部世界。面板上的按钮就好像是公共成员函数，人人都可用。但是复杂的电路则装配在机壳内部，就好像是保护成员，与外部隔离。protected 关键字就是电视机外壳。

(2) 使类对它本身内部实现的维护负责。因为只有类自己才能访问该类的保护数据，所以一切对保护数据的维护只有靠类自己了。如果其他的类或函数能够自由访问该类的保护成员，那么还让该类对它自己的内部实现负责是不公平的。如果某人打开了电视机外壳，改动了内部电路，使电视机遭损，要求厂家退换，厂家概不负责。但是，在使用外部按钮时，突然电视机发生故障，那厂家要负全面的责任，因为电视机是他们造的，内部电路的实现只有他们知道。

(3) 限制类与外部世界的接口。把一个类分作两部分，一部分是公共的，另一部分是保护的。保护成员对于使用者来说是不可见的，也是无须了解的。了解和使用一个具有有限接口(公共成员)的类是很容易的。电视机的按钮就寥寥几个，人们很容易学会使用，无须知道电视机的内部实现，照样可以将电视机用得好好的。

(4) 减少类与其他代码的关联程度。类的功能是独立的，它不依赖于应用程序的运行环境，而可以放在这个程序中使用，也可以放到那个程序中使用。这样，使得你能够非常容易地用一个类替换另一个类。电视机你会用，我也会用；放在你家可以播放，放在我家也可以播放，它并不要求我家具有一个特殊的电源或者专门的天线而工作。

类的保护机制使得人们编制的应用程序更加可靠和易维护。人们在编程时，误访问保护成员，编译就能报错来阻止基于错误的逻辑思考和编码。由于使用类的公共成员的错误，而导致程序运行不正常，会很快发现和纠正。当你购买了电视机后，图像不清楚，只要调一下按钮就行了；如果再调不好，那就去店里换吧，一定是电视机内部有问题。



假定一个学生类 Student,它具有 3 个功能:

增加课程:void AddCourse(int hours,float grade);

参数 hours 为课程学时,grade 为每学时成绩;

返回当前平均成绩:float Grade();

返回本学期学时数:int Hours()。

Student 的其余成员可声明为保护的,以便其他函数的操作与学生数据相隔离。类代码实现如下,我们用头文件来保存它:

```
//student.h

class Student
{
  public:
    float Grade()                                      //取当前平均成绩
    {
      return gpa;
    }
    int Hours()                                        //取学时数
    {
      return semesHours;
    }

    float AddCourse(int hours,float grade)             //增加课时及成绩
    {
      gpa = semesHours * gpa + grade * hours;          //总分
      semesHours  += hours;                            //调整学期学时数
      gpa / = semesHours;                              //调整平均成绩
    }
  protected:
    int semesHours;                                    //学期学时数
    float gpa;                                         //平均成绩
};
```

将这个类定义取名为 student.h,作为一个头文件保存。使用这个类时,只需将该头文件包含进来。

例如,下面的代码企图访问学生成绩:

```
#include "student.h"
#include <iostream>
using namespace std;
int main()
{
  Student s;

  //...
  s.gpa = 3.5;      //error
  cout << s.gpa << endl;      //error,但可以 s.Grade()
}
```

应用程序的编制者想提高某人平均成绩,又怕太高显得虚假,3.5 正好,所以“s.gpa=3.5;”,但是程序编译通不过,因为非法访问了保护数据成员 gpa。后面一句是同一错误。



他之所以无法修改他的程序来达到目的，是因为类的设计（头文件 student.h）中，没有单独修改保护数据的成员函数。Grade()和 Hours()都是取数据成员函数。

另一个学生管理员，他想维护学生成绩，编制了下面的应用程序：

```
#include "student.h"
#include <iostream>
using namespace std;
int main()
{
  Student s;
  //...

  cout << s.Grade()<< endl;
  cout << s.Hours()<< endl;
  float gpa = s.Grade();
  int hours = s.Hours();
  gpa += 3;                //修改局部变量 gpa 并不能使 s 对象中数据得以修改
  hours += 4.0;
  cout << s.gpa << endl;   //error
  cout << s.hours << endl;
}
```

他想看一下该学生原来的学分与成绩，然后给他加一门课程的成绩，但是新建局部变量并不能代表 s 对象中的学分和成绩，他使用错了。所有这样那样的错误，都可以简单地查出。

若将上面的代码做如下修改，就可运行得很好，即：

```
#include "student.h"
#include <iostream>
using namespace std;
int main()
{
  Student s;
  //...

  cout << s.Grade()<< endl;
  cout << s.Hours()<< endl;

  s.AddCourse(3,4.0);          //通过合法途径修改学时数与平均成绩

  cout << s.Grade() << endl;
  cout << s.Hours() << endl;
}
```

应该养成类编制的书写习惯。类定义中总是以 public、protected 或 private 开始，让人一目了然。

可以把类的成员声明作为私有的（private），使外部不能访问它们而起到保护作用。一个类定义，如果不写访问控制说明符（public、protected、private），那么它就默认为 private。

例如，下面的代码没有访问控制说明符：




```
class A
{
  int x;
  int y;
}
```

等价于：

```
class A
{
  private:
    int x;
    int y;
}
```

protected 和 private 的区别，在类的继承中才表现出来，详见 17.5 节。

11.6 屏蔽类的内部实现

类能够保护它的内部状态。在 11.5 节中，把数据成员 gpa 声明为保护的，以防止在应用程序中对平均成绩乱赋值。AddCourse()增加课程的成绩，会使 gpa 发生变化，但不能单独修改 gpa。如果要求有直接修改 gpa 的操作，则类能提供一个成员函数来达到这个目的。

例如，下面的代码在 11.5 节建立的学生类定义(student.h)中添加了单独修改平均成绩的成员函数 Grade(float)：

```
class Student
{
  public:
    float Grade()
    {
      return gpa;
    }

    float Grade(float newgpa)       //修改平均成绩
    {
      float oldgpa = gpa;
      if(newgpa >= 0&&newgpa <= 5.0)
        gpa = newgpa;

      return oldgpa;
    }
    //...
  protected:
    int semesHours;
    float gpa;
};
```

尽管允许修改成绩，但不是直接访问 gpa，而是告诉成员函数，由成员函数有防备地改。成员函数先审查要赋的值是否合理(在[0,5.0]范围内)，如果不符合要求，则仅将原值返回。返回原值的目的是防止误操作覆盖原值而使数据丢失。Grade(float)是重载函数。

这就是访问保护数据成员的途径，Student 类补充公共的成员函数去访问保护数据，在

公共成员函数中，可以施加种种操作的限制条件，以此对保护数据成员取值范围加以保护。如果以后发现 gpa 数据成员的值不正常，只要查看能改变其值的两个成员函数 Grade(float)和 AddCourse()即可。

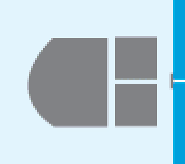
编制应用程序，想要使用某个类，所要了解的全部内容是它的公共成员，它们有什么用，参数是什么。我们使用电视机，只要学会几个按钮的用法就可以了，并不需要了解复杂的内部构造原理。

通过限制接口，很容易使用类。

由于条件的改变，或者发现了类中的错误，则只希望改变类的内部代码，而并不要求改变外部应用，因为接口没有变。

例如，下面的程序实现了一个 Point 类，并使用该类计算点的直角坐标和极坐标：

```
//----------------------
//    ch11_5.cpp
//----------------------
#include <iostream>
#include <cmath>
using namespace std;
//----------------------
class Point{
public:
  void Set(double ix,double iy){      //设置坐标
    x = ix; y = iy;
  }
  double xOffset(){                   //取 y 轴坐标分量
    return x;
  }
  double yOffset(){                   //取 x 轴坐标分量
    return y;
  }
  double angle(){                     //取点的极坐标
    return (180/3.14159) * atan2(y,x);
  }
  double radius(){                    //取点的极坐标半径
    return sqrt(x * x + y * y);
  }
protected:
  double x;                           //x 轴分量
  double y;                           //y 轴分量
}; //--------------------
bool getInput(double& x, double& y){
  cout <<"Enter x and y:\n";
  cin >> x >> y;
  return x >= 0;
}//----------------------
int main(){
  Point p;
  for(double x,y; getInput(x,y); ){ //输入 x,y 分量,直到 x<0
    p.Set(x,y);
    cout <<"angle = "<< p.angle()
        <<",radius = "<< p.radius()
```






```
            <<",x offset = "<< p.xOffset()
            <<",y offset = "<< p.yOffset()<< endl;
    }
}//--------------------
```

运行结果为：

```
Enter x and y:
10 10
angle = 45,radius = 14.1421,x offset = 10,y offset = 10
Enter x and y:
50 0
angle = 0,radius = 50,x offset = 50,y offset = 0
Enter x and y:
-1 -1
```

类中，保护数据存放 Point 对象的 x、y 坐标，一切对 x、y 的操作都由成员函数来完成。Set()设置 x、y 坐标，xOffset()、yOffset()分别返回 x、y 的值，angle()、radius()分别返回 x、y 极坐标的值。

math.h 是 C 库函数头文件，专门描述数学函数，而 cmath 则是对应 C++的数学函数头文件，其为 math.h 的改造版，使之适应 C++编程，例如允许函数重载等。此处包含 cmath 和 math.h 等价。反正切函数 atan()以弧度为参数(此处没有用到)，另一个反正切函数 atan2()则以 x、y 的坐标分量做参数，二者是不同的。此处用到的函数原型如下：

```
double sin(double x);               //x 为弧度,求其正弦值
double cos(double x);               //x 为弧度,求其余弦值
double atan2(double y,double x);    //y、x 为坐标分量,求 y/x 的反正切值
double sqrt(double x);              //求平方根
```

将 Point 类从程序中分离，成为独立的头文件 point.h，则程序可以改写为如下：

```
//---------------------
//     ch11_6.cpp
//---------------------
#include"point.h"
#include<iostream>
using namespace std;
//---------------------
bool getInput(double& x, double& y){
  cout<<"Enter x and y:\n";
  cin>>x>>y;
  return x>=0;
}//--------------------
int main(){
  Point p;
  for(double x,y; getInput(x,y); ){ //输入 x、y 分量,直到 x<0
    p.Set(x,y);
    cout<<"angle = "<<p.angle()
        <<",radius = "<<p.radius()
        <<",x offset = "<<p.xOffset()
        <<",y offset = "<<p.yOffset()<<endl; }
}//--------------------
```

该程序与 ch11_5.cpp 的功能完全一样。其中的头文件为:

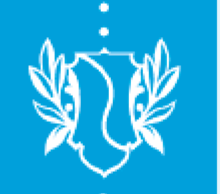
```
//----------------------
//     point.h
//----------------------
#include <cmath>                    //用到 atan2()、sqrt()
using namespace std;
//----------------------
class Point{
public:
  void Set(double ix,double iy){  //接口
    x = ix; y = iy;
  }
  double xOffset(){               //接口
    return x;
  }
  double yOffset(){               //接口
    return y;
  }
  double angle(){                 //接口
    return (180/3.14159) * atan2(y,x);
  }
  double radius(){                //接口
    return sqrt(x * x + y * y);
  }
protected:
  double x;
  double y;
}; //--------------------
```

由于头文件中要用到数学函数,所以 point.h 中包含 cmath 头文件。

由于某种原因,类要修改,但接口不变,也就是说,公共成员函数名字、功能、参数形式不变,那么用户的应用程序就无须改变,即 ch11_6.cpp 无须作任何修改。

下面的 Point 类修改了保护数据结构。类的保护数据成员改为点的极坐标形式 a 和 r,相应地,成员函数的实现也要作修改。具体如下:

```
//----------------------
//     point.h
//----------------------
#include <cmath>                    //用到 atan2()、sqrt()、sin()、cos()
using namespace std;
//----------------------
class Point{
public:
  void Set(double ix,double iy){  //接口
    a = atan2(iy,ix);
    r = sqrt(ix * ix + iy * iy);
  }
  double xOffset(){               //接口
    return r * cos(a);
  }
  double yOffset(){               //接口
    return r * sin(a);
  }
```




```
    double angle(){          //接口
      return (180/3.14159) * a;
    }
    double radius(){         //接口
      return r;
    }
  protected:
    double a;
    double r;
  }; //--------------------
```

类的内部实现改变了,即保护数据成员结构和成员函数内部算法都不一样了,但接口没做任何改变,所以 ch11_6.cpp 程序也无须作改动,运行结果不变。

由于类很好地屏蔽了内部数据表示,所以由类负责的内部实现上的维护不影响应用程序的开发,类编程与应用编程作了分工,职责明确,使得编程工作可以模块化运作,这大大减轻了开发应用程序的强度。

11.7 名字识别

1. 类的作用域

一个类的所有成员位于这个类的作用域内,一个类的任何成员都能访问同一类的任一其他成员。C++认为一个类的全部成员都是一个整体的相关部分。

例如,在 11.5 节中的 Student.h 头文件对 Student 类的成员函数定义中,AddCourse()成员函数能够访问类中数据成员 semesHours 和 gpa。

类作用域是指类定义和相应的成员函数定义范围。在该范围内,一个类的成员函数对同一类的数据成员具有无限制的访问权。

对类作用域外的一个类的数据成员和成员函数的访问受到访问权限的制约。这种思想是要把一个类的数据结构和功能封装起来,从而使得在类的成员函数之外对类的数据访问加上限制。

例如,在 Student 类中,保护数据 gpa 是不能让普通函数直接访问的,见 11.5 节。

2. 可见性

两个名字在同一作用域内,由于层次不同,内层名字往往会遮挡外层名字。例如,m 是 X 类的数据成员,且其成员函数定义中有同名的局部作用域变量,则:

```
 class X
{
  public:
    void f1();
    void f2();
  protected:
    int m;
};

void X::f1()
```

```
{
  m = 5;
}

void X::f2()
{
  int m;
  m = 2;     //X::m 被隐藏
}
```

类 X 的数据成员 m 的作用域尽管在类 X 中，但是成员函数中定义了同名的局部作用域变量后，就把数据成员 m 给隐藏了。

3. 类名遮挡

类名允许与其他变量名或函数名相同。当类名与程序中的其他变量或者函数名相同时，可以通过下面的方法来正确访问。

（1）如果一个非类型名隐藏了类型名，则类型名通过加前缀可用。

例如，一个类名被在后面的函数中的形参所覆盖，在该函数内，要定义一个类对象，则加上 class 即可：

```
class Sample                  //定义类
{
   //...
};

void func(int Sample)         //函数形参隐藏了类名
{
  class Sample a;             //定义一个对象要用到类名
  Sample ++ ;                 //形参的算术操作
  //...
}
```

（2）如果一个类型名隐藏了一个非类型名，则用一般作用域规则即可。

例如：

```
int s = 0;                 //全局变量
void func()
{
   class s{  //... };      //类 s 隐藏了全局变量 s
   s a;                    //定义一个类对象
   ::s = 3;                //引用全局变量
}                          //class s 作用域结束
int g = s;                 //用全局变量 s 给变量 g 初始化
```

→ 在函数中定义的类称为局部类，局部类在面向对象程序设计中并不多见。类作为类型也有作用域，局部类的作用域在定义该类的函数块中。

局部类的成员函数必须在类定义内部定义，因为若在类外部和包含该类的函数内部中定义，则导致在函数内部定义函数的矛盾。如果在包含类的函数外部定义，则该局部类无法与其取得联系。



4. 名空间

名空间是指某名字在其中必须唯一的作用域。

C++规定：一个名字不能同时指两种类型。例如：

```
class C                          //定义一个类类型
{
  //...
};
typedef int C;                   //error: 又定义一个类型取同名
```

非类型名(变量名、常量名、函数名、对象名或枚举成员)不能重名。例如：

```
Student a;                       //定义一个对象
void a();                        //error: 函数名与对象名同名
```

类型与非类型不在同一名空间。也即在一个作用域中,一个名字可以声明为一个类型,也可以声明为一个非类型。当二者同时登场时,类型名要加前缀,以区别非类型名。例如：

```
class stat                       //先定义类类型
{
  //...
};
stat a;                          //定义类对象
void stat(stat * ps);            //函数名与类名相同,它们不在同一名空间
class stat b;                    //必须区分 stat 是类型还是函数
stat(0);                         //非类型名: 函数调用
```

又如：

```
int f(int);                      //先定义一个非类型名
class f                          //定义一个类
{
  //...
};
class f g;                       //必须加 class 以区分 f 是类型名
```

又如：

```
class A
{
  //...
};
A A;        //定义一个 A 类型的对象 A,合法但不可取
```

11.8 再论程序结构

1. 类的封装

类的封装的概念首先是,数据与算法(操作)结合,构成一个不可分割的整体(对象)。其



次是，在这个整体中一些成员是保护的，它们被有效地屏蔽，以防外界的干扰和误操作；另一些成员是公共的，它们作为接口提供给外界使用。而对该对象的描述即是创建用户定义的类型，对 C++来说，就是类的实现机制。

图 11-4 是一个 Point 类的描述。这是具有一般意义的，许多有关面向对象的书都采用这种形式的描述。

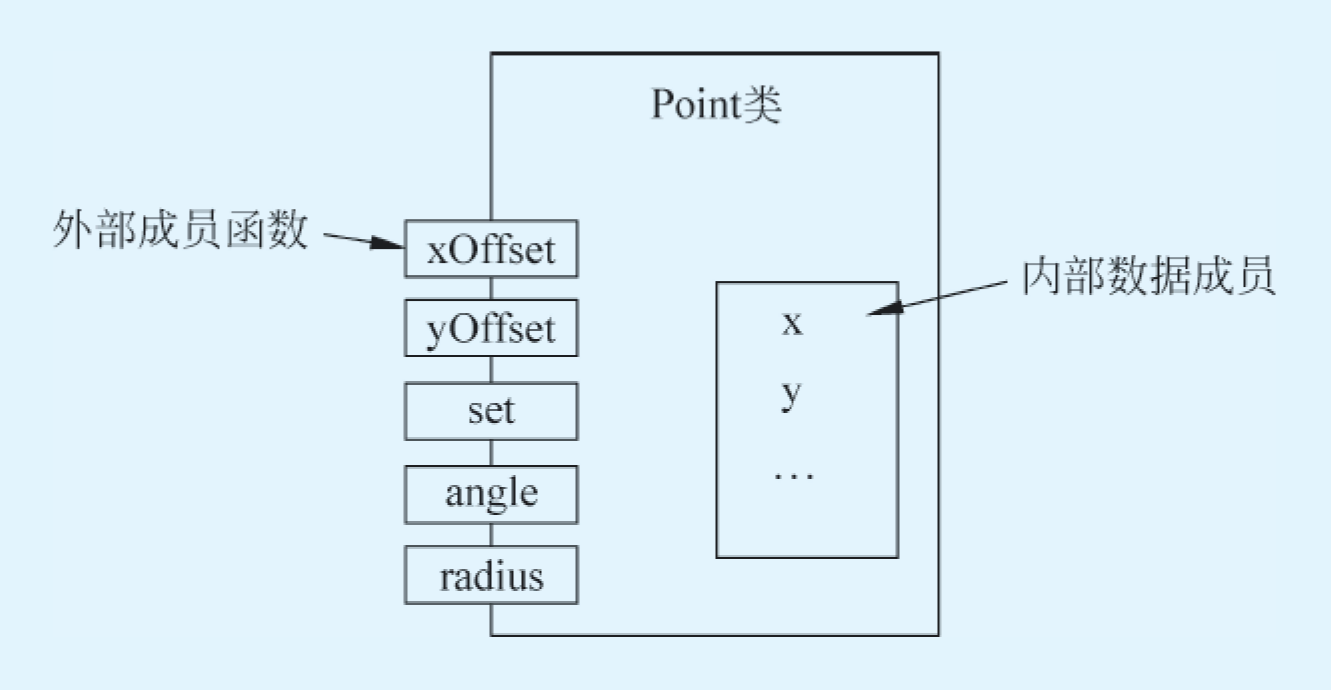


图 11-4　Point 类的描述

把图 11-4 用 C++语言来表达，就是 11.6 节的头文件 point.h 中的类定义和相关的成员函数定义。将类的描述分成类的外部接口和类的内部实现就是下面两个文件的代码：

```
 point.h              //类的外部接口,修改了的头文件

class Point
{
  public:             //外部接口
    void Set(double ix,double iy);
    double xOffset();
    double yOffset();
    double angle();
    double radius();
  protected:          //内部数据
    double x;
    double y;
    };

point.cpp             //类的内部实现

#include "point.h"
#include <cmath>
using namespace std;
void Point::Set(double ix,double iy)
{
  x = ix;
  y = iy;
}
double Point::xOffset()
{
  return x;
}
double Point::yOffset()
```




```
    {
      return y;
    }
    double Point::angle()
    {
      return (180/3.14159) * atan2(y,x);
    }
    double Point::radius()
    {
      return sqrt(x * x + y * y);
    }
```

在 point.h 头文件中,是一个类定义。其中保护数据成员并不是外部接口部分,但作为类定义的整体(从开括号起到闭括号止)描述,只好把它放在头文件中。对于面向对象程序设计之严格的内外界面描述来说,这是 C++类机制中无可奈何的一面。但它不妨碍类的外部接口的有效性。在 11.6 节屏蔽类的内部实现中我们看到了这一点。

在 point.cpp 文件中,每个成员函数都加了类名,在文件开始包含了两个头文件,一个是 point.h,这是必要的,因为所有成员函数的声明都在类定义中,这就好像标准函数头文件的结构一样;另一个是 cmath,因为成员函数定义中调用了 cmath 中声明的函数。

2. C++程序结构

一个 C++应用程序是一个程序工程。一个 C++程序工程文件中,应该组合下面这些程序文件:

```
main.cpp           //包含主函数的程序文件
class.cpp's        //用户自定义类库的内部实现程序
function.cpp's     //用户自定义函数库的实现程序
```

其中,class.cpp's 表示多个类成员函数定义的源文件;function.cpp's 表示多个函数定义的源文件。

其中,包含主函数的源文件应该是下面的形式:

```
main.cpp 程序文件

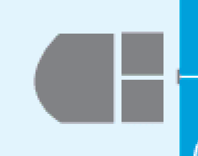
#include <标准类库头文件>'s
#include <标准函数头文件>'s
#include "自定义类库头文件"'s
#include "自定义函数头文件"'s

函数原型's
全局数据定义's

int main()
{
  //...
}

函数定义's
```

这里包含标准类库头文件,即称为类库重用,包括一个类定义和成员函数定义。类定义



以头文件的方式提供,成员函数定义则以一定的计算机硬件或操作系统为背景而编译实现的内部代码方式提供。

自定义类库头文件称为类库设计。而在 main()函数开始之后的面向对象程序设计,则显得相对自然和简洁。往往是先定义若干对象,然后调用其成员函数,由成员函数来完成程序员所规定的操作(即让对象表现自己)。

小结

一个类具有数据成员,还具有成员函数,通过成员函数可以对数据成员进行操作,并实现其他的功能。

定义了一个类后,可以把该类名作为一种数据类型,定义其“变量”(对象)。

程序利用点操作符(.)访问类的公共成员。

程序可以在类的外部或内部定义它的成员函数,在类的外部定义成员函数时,必须指出所属的类名,并用全局作用域分辨符(::)把类名和函数名连接起来。

类的成员,包括数据和函数,都可以被说明为公有、保护或私有。公有成员可以在程序中任意被访问,而保护或私有成员只能被这个类的成员函数所访问。

把成员说明为保护的,使类的使用者在使用它时只关心接口,无须关心它的内部实现,既方便了使用,又保护了内部结构。这就是类的封装原理。

含有类的程序结构充分体现了类的封装和重用,更容易被人所理解。

练习

11.1　下列程序有几个错误?

```
#include <iostream>
#include <cmath>
using namespace std;
class Point
{
  public:
    void Set(double ix,double iy)  //设置坐标
    {
      x = ix;
      y = iy;
    }

    double xOffset()               //取 y 轴坐标分量
    {
      return x;
    }

    double yOffset()               //取 x 轴坐标分量
    {
      return y;
    }
    double angle()                 //取点的极坐标 θ
```




```
        {
          return (180/3.14159) * atan2(y,x);
        }

        double radius()        //取点的极坐标半径
        {
          return sqrt(x * x + y * y);
        }
    protected:
        double x;              //x 轴分量
        double y;              //y 轴分量
}

int main()
{
    Point p;
    double x,y;

    cout <<"Enter x and y:\n";
    cin >> x >> y;

    p.Set(x,y);
    p.x += 5;
    p.y += 6;
    cout <<"angle = " << p.angle()
         <<",radius = " << p.radius()
         <<",x offset = " << p.xOffset()
         <<",y offset = " << p.yOffset() <<"endl;
}
```

11.2　将下列程序分离类定义和主函数,改成多文件结构。主函数使用类的方式采取包含类定义的头文件的方法。写出运行结果。

```
#include <iostream>
using namespace std;
class Cat
{
    public:
        int GetAge();
        void SetAge(int age);
        void Meow();       //喵喵叫
    protected:
        int itsAge;
};

int Cat::GetAge()
{
    return itsAge;
}

void Cat::SetAge(int age)
{
    itsAge = age;
}

void Cat::Meow()
```

```
{
  cout <<"Meow.\n";
}

int main()
{
  Cat frisky;
  frisky.SetAge(5);
  frisky.Meow();
  cout <<"frisky is a cat who is "
      << frisky.GetAge()
      <<" years old.\n";
  frisky.Meow();
}
```

11.3　定义一个满足如下要求的 Date 类。

(1) 用下面的格式输出日期：

```
日/月/年
```

(2) 可运行在日期上加一天操作；

(3) 设置日期。

11.4　定义一个时间类 Time，能提供和设置由时、分、秒组成的时间，并编写应用程序，定义时间对象，设置时间，输出该对象提供的时间。并请将类定义作为接口，用多文件结构实现之。

11.5　编写一个类，实现简单的栈（提示：用链表结构实现）。数据的操作按先进后出（FILO）的顺序。

成员函数为：

```
void queue::put(int item);     //将数据 item 插入栈中
int queue::get();              //从栈中取一个数据
```

数据成员为：

一个指向链首的指针

链表结构为：

```
struct Node
{
  int a;
  Node * next;
};
```

使用对象的过程：

```
queue que;

que.put(10);
que.put(12);
que.put(14);

cout << que.get() << endl;    //输出 14,栈中剩下 10、12
cout << que.get() << endl;    //输出 12,栈中剩下 10
```

第12章　构造函数

C++的构造函数和析构函数使类对象能够轻灵地被创建和撤销。构造函数创建类对象，初始化其成员；析构函数撤销类对象。构造函数和析构函数是类的特殊成员函数，它们的设计与应用直接影响编译程序处理对象的方式。构造函数和析构函数的实现使C++的类机制得以充分地展示。所以本章内容是C++的重点之一。学习本章后，要求理解类与对象的区别，掌握定义构造函数和析构函数的方法，把握默认构造函数的意义，了解类成员初始化的问题，掌握构造类成员的方法。

12.1　类与对象

1. 类与对象的区别

人类是一个类，你是人，我是人，都是人类的实例(instance)，或称对象(object)。一个类描述一类事物，描述这些事物所应共同具有的属性，如人有身高、体重、文化程度、性别、年龄、民族等。

一个对象是类的一个实例，它具有确定的属性，如张三(人的实例)身高180cm，体重70kg，大学本科，男，21岁，汉族。

人类只有一个，人类的实例可以有无数个。

对象可以被创建和销毁，但类是无所不在的。

例如，桌子是一个类，人们不断打造各种尺寸和风格(属性)的桌子(桌子的实例)，又不断毁坏桌子，年复一年，旧的去了，新的又来，但桌子的概念没变，它是一个抽象的概念。应该称它为桌子类，以区别于打造的具体桌子。

2. 定义对象

属于不同类的对象在不同的时刻、不同的地方分别被建立。全局对象在主函数开始执行前先被建立，局部对象在程序执行遇到它们的对象定义时才被建立。与定义变量类似，定义对象时，C++为其分配内存空间。



例如，下面的代码定义了两个类，创建了类的全局对象、局部对象、静态对象和堆对象：

```
class Desk                 //Desk 类
{
  public:
    int weight;
    int high;
    int width;
    int length;
};

class Stool                //另一个类:Stool 类
{
  public:
    int weight;
    int high;
    int width;
    int length;
};

Desk da;                   //全局对象
Stool sa;

void fn()
{
  static Stool ss;         //静态局部对象
  Desk da;                 //局部对象
  //...
}

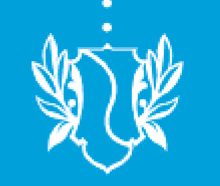
int main()
{
  Stool bs;                //局部对象
  Desk * pd = new Desk;    //堆对象
  Desk nd[50];             //局部对象数组
  //...
  delete pd;               //释放堆对象
}
```

3. 对象的初始化

根据变量定义，全局变量和静态变量在定义(分配空间)时，将位模式清 0；局部变量在定义时，分配的内存空间内容保持原样，故为随机数。

对象定义时，情况不一样。对象的意义表达了现实世界的实体，因此，一旦建立对象，需要有一个有意义的初始值。C++建立和初始化对象的过程专门由该类的构造函数来完成。这个构造函数很特殊，只要对象建立，它马上被调用，给对象分配内存空间和初始化。例如一旦打造了一张桌子，桌子就应有长、宽、高和重量。因此，在桌子对象建立时，构造函数的任务是给该桌子对象赋予一组初始值。如果一个类没有专门定义构造函数，那么 C++就仅仅创建对象而不做任何初始化。

C++另有一种析构函数，它也是类的成员函数，当对象撤销时，就会马上被调用，其作用



是释放与对象捆绑的资源,做一些善后处理。例如,一张桌子要扔掉,需要将桌子里面的东西拿出来,这些东西可能有用,不能随桌子一起扔。类似这些事就由析构函数来完成。

12.2 构造函数的必要性

变量初始化的方法如下所示:

```
int a = 1; int * pa = &a;
```

数组初始化的方法如下所示:

```
int b[ ] = {1,2,3,4};
```

结构初始化的方法如下所示:

```
struct Student
{
  int semesHours;          //总需学时数
  float gpa;               //平均成绩
};

void fn()
{
  Student s = {100, 3.5}; //创建结构变量时,初始化
  //...
}
```

但是对类对象来说,如此初始化不行,这是由类的特殊性所决定的。

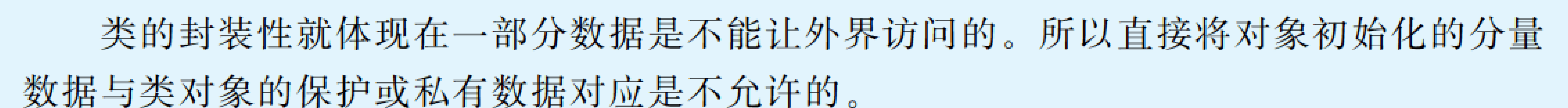
例如,下面的代码企图在对象创建时,为其初始化:

```
class Student
{
  public:
  //...公共成员...
  protected:
    int semesHours;          //此处的数据成员是受保护的
    float gpa;
};

void fn()
{
  Student s = {100, 3.5};    //error: 不能访问
  //...
}
```

如果上面的函数允许那样初始化的话,就意味着在函数中允许访问类对象中的保护数据成员:

```
void fn()
{
  s.semesHours = 0;
  s.gpa = 0;
}
```


类的封装性就体现在一部分数据是不能让外界访问的。所以直接将对象初始化的分量数据与类对象的保护或私有数据对应是不允许的。

类对象的初始化任务自然就落在了类的成员函数身上，因为它们可以访问保护和私有数据成员。于是将初始化构想成下面的形式：

```
class Student
{
  public:
    void init()
    {
      semesHours = 100;
      gpa = 3.5;
    }
    //...其他公共成员
  protected:
    int semesHours;
    int gpa;
};

void fn()
{
  Student s;
  s.init();     //类的初始化
  //函数的其他部分
}
```

类的保护数据在外界是不能访问的，所以对这些数据的维护是类的份内工作。将初始化工作交由 init()成员函数无可非议，但却让系统多了一道处理初始化的解释与执行的步骤，因为它意味着在编写应用程序中每当建立对象时，都要非统一地人为调用函数以修改对象成员。这样实现的类机制并不理想。

我们要求建立对象的同时自动调用构造函数，省去人为调用的麻烦。也就是说，类对象的定义：

```
Student ss;
```

明确表达了为对象分配空间和初始化的意向。这些工作由构造函数来完成。

类对象的定义“Student ss;”涉及一个类名和一个对象名。类只有一个名字，但可以有多个对象名，每个对象创建时，都要调用该类的构造函数。类的唯一性和对象的多样性，使我们马上想到用类名而不是对象名来作为构造函数名是比较合适的。

12.3 构造函数的使用

C++规定与类同名的成员函数是构造函数，在该类的对象创建时，自动被调用。

例如，下面的代码初始化桌子和凳子类对象：




```
class Desk
{
  public:
       Desk()          //构造函数定义
       {
         weight = 10;
         high = 5;
         width = 5;
         length = 5;
       }
  protected:
        int weight;
        int high;
        int width;
        int length;
};

class Stool
{
  public:
       Stool()         //构造函数定义
       {
         weight = 6;
         high = 3;
         width = 3;
         length = 3;
       }
  protected:
        int weight;
        int high;
        int width;
        int length;
};

void fn()
{
  Desk da;             //自动调用 Desk(),创建对象并初始化
  Stool sa;            //自动调用 Stool()
  //...
}
```

与成员函数相同,构造函数可以放在类的外部定义。

例如,下面的程序是将上例代码中的构造函数放在类的外部定义:

```
//----------------------
//    ch12_1.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class Desk{
public:
  Desk();                            //构造函数声明
private:
```

```
    int weight, high, width, length;
}; //--------------------
class Stool{
public:
    Stool();           //构造函数声明
private:
    int weight, high, width, length;
}; //--------------------
Desk::Desk(){          //构造函数定义
    weight=10; high=5; width=5; length=5;
    cout<<weight<<" "<<high<<" "<<width<<" "<<length<<endl; }
//--------------------
Stool::Stool(){        //构造函数定义
    weight=6; high=3; width=3; length=3;
    cout<<weight<<" "<<high<<" "<<width<<" "<<length<<endl; }
//--------------------
void fn(){
    Desk da;           //自动调用 Desk()
    Stool sa;          //自动调用 Stool()
}//--------------------
int main(){
    fn();
}//--------------------
```

运行结果为：

```
10 5 5 5
6 3 3 3
```

主函数 main()开始运行时，调用 fn()函数，fn()函数在创建 Desk 对象 da 和 Stool 对象 sa 时，分别调用了二者的构造函数。它们在内存中的空间分配如图 12-1 所示。da 和 sa 对象空间中，依次存放着 weight、high、width 和 length 的值。

da	10
	5
	5
	5
sa	6
	3
	3
	3

图 12-1　桌子凳子对象在内存中的分配

放在外部定义的构造函数，其函数名前要加上“类名::”，这和别的成员函数定义方法一样。因为在类定义的外部可能有各种函数定义，为了区分成员与非成员函数，区分此类成员函数和彼类成员函数，所以加上“类名::”是必要的。

构造函数另一个特殊之处是它没有返回类型，函数体中也不允许返回值，但可以有无值返回语句“return;”。因为构造函数专门用于创建对象并为其初始化，所以它不能随意被调用。没有返回类型，正显得它与众不同。

例如,下面的代码是在类定义的外部定义一个构造函数,但是却错误地加上了返回类型:

```
class Desk
{
  public :
        Desk();     //构造函数声明
  protected:
        int weight;
        int high;
        int width;
        int length;
};

void Desk::Desk()    //error: 不能有返回类型
{
  weight = 10;
  high = 5;
  width = 5;
  length = 5;
}
```

如果在 ch12_1.cpp 的函数 fn()中,定义桌子对象的语句改成定义对象数组:

```
void fn()
{
  Desk dd[5];     //对象数组 dd
  Stool sa;
  //...
}
```

则定义对象数组的语句会调用 5 次构造函数。因为每个元素都是一个对象,每创建一个对象都会调用构造函数。从 dd[0]开始到 dd[4],逐个创建对象并初始化。所以 ch12_1.cpp 代码若修改一下,输出全部对象,则输出结果为:

```
10 5 5 5
10 5 5 5
10 5 5 5
10 5 5 5
10 5 5 5
6 3 3 3
```

一个类定义中,类的数据成员可能为另一个类的对象。

例如,下面代码的类结构表示在组合音响中包含的各个套件类对象:

```
class Recorder
{
  //...
};
class Cdplayer
{
  //...
};
class Amplifier
{
```

```
    //...
};
class Tuner
{
    //...
};
class HiFi
{
    public:
    //...
    protected:
          Recorder re;
          Cdplayer cd;
          Amplifier am;
          Tuner tu;
};
```

如果一个类对象是另一个类的数据成员，则在创建那个类的对象所调用的构造函数中，对该成员(对象)自动调用其构造函数。

例如，在下面程序中，“帮教派对”类 TutorPair 包含学生类对象和老师类对象：

```
//----------------------
//    ch12_2.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class Student{
public:
  Student(){
    cout <<"constructing student.\n";
    semesHours = 100;
    gpa = 3.5;
  }
  //其他公共成员
private:
  int semesHours;
  float gpa;
}; //--------------------
class Teacher{
public:
  Teacher() {
    cout <<"constructing teacher.\n";
  }
}; //--------------------
class TutorPair{
public:
  TutorPair(){
    cout <<"constructing tutorpair.\n";
    noMeetings = 0;
  }
private:
  Student student;
```




```
    Teacher teacher;
    int noMeetings;      //会晤次数
};//--------------------
int main(){
    TutorPair tp;
    cout <<"back in main.\n";
}//---------------------
```

运行结果为:

```
constructing student.
constructing teacher.
constructing tutorpair.
back in main.
```

主函数 main()运行开始时,遇到要创建 TutorPair 类的对象,于是调用其构造函数 TutorPair(),该构造启动时,首先分配对象空间(包含一个 Student 对象、一个 Teacher 对象和一个 int 型数据),然后根据在类中声明的对象成员的次序依次调用其构造函数。这里先调用 Student()构造函数,再调用 Teacher()构造函数,最后才执行它自己的构造函数的函数体。按照这个顺序,分别产生运行结果的第一、第二、第三行输出。当执行完 TutorPair()构造函数后,控制权回到主函数中,产生第四行输出。

这个例子告诉我们,类在工作过程中分工十分明确,每个类只负责初始化它自己的对象。当 TutorPair 类要初始化成员 Student 类对象的时候,马上调用 Student 构造函数,而不是由自己去包办,正所谓"你做你的事,我做我的事",这在 12.8 节和 12.9 节中将作进一步介绍。

12.4 析构函数

一个类可能在构造函数里分配资源,这些资源需要在对象不复存在前被释放。例如,如果构造函数打开了一个文件,文件就需要被关闭;或者,如果构造函数从堆中分配了内存,这块内存在对象消失之前必须被释放。析构函数允许类自动完成这些清理工作,不必调用其他成员函数。堆对象析构的内容在 14.3 节进一步描述。

析构函数也是特殊的类成员函数,它没有返回类型,没有参数,不能随意调用,也没有重载。它只是在类对象生命期结束的时候,由系统自动调用。在 12.5 节和 12.6 节将会看到,构造函数不同于析构函数,它可以有参数,可以重载。

作为一个类,可能有许多对象,每当对象生命期结束时,都要调用析构函数,每个对象一次。这跟构造函数形成了鲜明的对立,所以析构函数名就在构造函数名前加上一个逻辑非运算符"~",表示"逆构造函数"。

例如,下面的代码定义了一个析构函数:

```
class XYZ
{
  public:
    XYZ()
    {
      name = new char[20];   //分配堆空间
```

```
    }
    ~XYZ()
    {
      delete name;   //释放堆空间
    }
    protected:
      char * name;
}
```

XYZ类的构造函数中分配了一段堆内存给作为指针的name数据成员。一旦对象创建,该对象就在对象空间之外拥有了一段堆内存资源。对应地,当对象在撤销的时候,首先必须归还这一堆内存资源。这个工作由析构函数做。

当你进入图书馆阅览室借书阅览时,你就成了一个阅览室的阅览人(对象),借什么书是由一进去就完成的(构造)。当你要离开阅览室(撤销对象)时,你必须先归还图书(析构)才能顺利地离去。

析构函数以调用构造函数相反的顺序被调用。

例如,下面的程序在ch12_2.cpp的基础上增加了析构函数:

```
//----------------------
//    ch12_3.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class Student{
public:
  Student(){
    cout <<"constructing student.\n";
    semesHours = 100;
    gpa = 3.5;
  }
 ~Student(){
    cout <<"destructing student.\n";
  }
  //其他公共成员
private:
  int semesHours;
  float gpa;
}; //-------------------
class Teacher{
public:
  Teacher(){
    cout <<"constructing teacher.\n";
  }
 ~Teacher(){
    cout <<"destructing teacher.\n";
  }
}; //-------------------
class TutorPair{
public:
  TutorPair(){
    cout <<"constructing tutorpair.\n";
```




```
        noMeetings = 0;
    }
  ~TutorPair(){
        cout <<"destructing tutorpair.\n";
    }
  private:
    Student student;
    Teacher teacher;
    int noMeetings;
  }; //-------------------
  int main(){
    TutorPair tp;
    cout <<"back in main.\n";
  }//---------------------
```

运行结果为：

```
constructing student.
constructing teacher.
constructing tutorpair.
back in main.
destructing tutorpair.
destructing teacher.
destructing student.
```

当主函数运行到结束的大括号处时，析构函数依次被调用，其调用顺序正好与构造函数相反。

12.5 带参数的构造函数

前面介绍的构造函数不能完全满足类对象初始化的要求。应该让构造函数可以带参数，否则，往往程序员只能先将对象构造成千篇一律的对象值，甚至构造一个随机值对象，然后再调用一个初始化成员函数将数据存到该对象中去。

例如，下面的代码构造一个人类的对象：

```
class Mankind
{
  public:
  //...
  protected:
    int age;
    int sex;
    Date birthday;
};

Mankind a;
```

在该人类的定义中，构造函数是默认的，所以创建的全局对象，其初始值全为0。对于一个人，0岁，无性别，无出生年月，是无意义的。创建对象应该是构造一个活生生的人。

带参数的构造函数使初始化一步到位。

例如，下面的程序定义了一个带参数的学生类：

```
//----------------------
//    ch12_4.cpp
//----------------------
#include <iostream>
#include <cstring> //用到 strncpy()
using namespace std;
//----------------------
class Student{
public:
  Student(char * pName){
    cout <<"constructing student "<< pName << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0'; //防止名字过长而崩溃
  }
~Student(){
    cout <<"destructing "<< name << endl;
  }
  //其他公共成员

private:
  char name[20];
}; //--------------------
int main(){
  Student ss("Jenny");
}//---------------------
```

运行结果为：

```
constructing student Jenny
destructing Jenny
```

学生类中定义了一个带字符串参数的构造函数。在主函数运行时，调用了 Student()构造函数，创建对象 ss，在这当中，输出一行信息，将参数 Jenny 复制给对象 ss，并在对象的数据成员 name 数组的最后一个元素 name[19]填上'\0'。

构造函数在参数规定上和普通函数一样，可以有任意多个参数。

例如，下面的程序修改了程序 ch12_4.cpp 中的学生类，含有 3 个参数：

```
//----------------------
//    ch12_5.cpp
//----------------------
#include <iostream>
#include <cstring> //用到 strncpy()
using namespace std;
//----------------------
class Student{
public:
  Student(char * pName, int xHours, float xgpa){
    cout <<"constructing student "<< pName << endl;
    strncpy(name,pName,sizeof(name));
    name[sizeof(name) - 1] = '\0';
    semesHours = xHours;
    gpa = xgpa;
  }
~Student(){
```




```
      cout <<"destructing "<< name << endl;
    }
    //其他公共成员
private:
    char name[20];
    int semesHours;
    float gpa;
}; //-------------------
int main(){
    Student ss("Jenny",22,3.5);
}//--------------------
```

运行结果为：

```
constructing student Jenny
destructing Jenny
```

该程序运行结果与 ch12_4.cpp 完全一样，但是，Student 对象空间不一样了。由于增加了数据成员，为初始化它们，构造函数带了 3 个参数，在主函数中，创建对象的语句要求对象名 ss 相应地也包括 3 个参数。它所创建的学生名叫 Jenny，学期学时数为 22，平均成绩 3.5。

12.6 重载构造函数

构造函数可以被重载，C++根据声明中的参数选择合适的构造函数。

例如，下面的程序同时声明了 4 个构造函数：

```
//---------------------
//    ch12_6.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Tdate{
public:
    Tdate();
    Tdate(int d);
    Tdate(int m, int d);
    Tdate(int m, int d, int y);
    //其他公共成员
private:
    int month, day, year;
}; //-------------------
Tdate::Tdate(){
    month = 4; day = 15; year = 1995;
    cout << month <<"/"<< day <<"/"<< year << endl;
}//--------------------
Tdate::Tdate(int d){
    month = 4; day = d; year = 1996;
    cout << month <<"/"<< day <<"/"<< year << endl;
}//--------------------
Tdate::Tdate(int m, int d){
    month = m; day = d; year = 1997;
```

```
    cout << month <<"/"<< day <<"/"<< year << endl;
  }//--------------------
  Tdate::Tdate(int m,int d,int y){
    month = m; day = d; year = y;
    cout << month <<"/"<< day <<"/"<< year << endl;
  }//--------------------
  int main(){
    Tdate aday;
    Tdate bday(10);
    Tdate cday(2,12);
    Tdate dday(1,2,1998);
  }//--------------------
```

运行结果为：

```
4/15/1995
4/10/1996
2/12/1997
1/2/1998
```

因为对象 aday 以无参数形式给出，所以用无参构造函数 Tdate()来构造它，无参的构造函数被称为默认构造函数。后面 3 个对象分别匹配另 3 个不同的构造函数。

由于构造函数用于创建对象，所以调用它来给对象赋值是错误的。

例如，下面的代码中，构造函数企图再次调用另一个重载的构造函数来简化编程：

```
Tdate::Tdate(int d)
{
  Tdate(4,d,1995);   //构造函数创建一个无名对象
}
```

显式调用构造函数将创建一个无名对象，关于无名对象，14.8 节将专门介绍。要想共享初始化的过程，可以先定义一个共享成员函数，然后每个构造函数都调用之。例如，下面的程序定义了一个名叫 Init 的成员函数：

```
//---------------------
//    ch12_7.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Tdate{
public:
  Tdate(){ Init(4,15,1995); }
  Tdate(int d){ Init(4,d,1996); }
  Tdate(int m,int d){ Init(m,d,1997); }
  Tdate(int m,int d,int y){ Init(m,d,y); }
  //其他公共成员
private:
  int month, day, year;
  void Init(int m,int d,int y){
    month = m; day = d; year = y;
    cout << month <<"/"<< day <<"/"<< year << endl;
  }
```




```
}; //--------------------
int main(){
  Tdate aday;
  Tdate bday(10);
  Tdate cday(2,12);
  Tdate dday(1,2,1998);
}//---------------------
```

运行结果为：

```
4/15/1995
4/10/1996
2/12/1997
1/2/1998
```

Tdate 类增加了一个做具体初始化工作的 Init()函数，由于它仅仅被用于构造函数的调用，即仅被类的成员函数调用，所以可以将其放在保护成员内，以阻止其他应用程序使用。

还可以通过给最后一个构造函数以参数默认值，例如设定最后一个构造函数的 3 个参数的默认值为 12、31、2003，那么参数不同的 4 种调用都能匹配该函数，从而使这 4 个构造函数结合成一个。

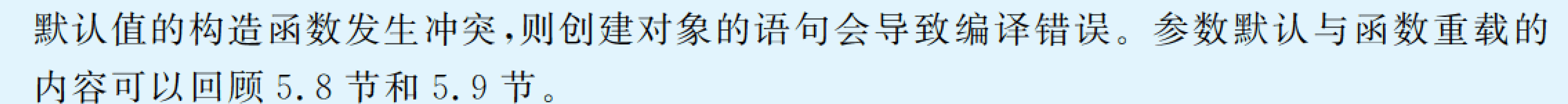
例如，下面的程序在类中，将 4 个构造函数结合为一个单一的构造函数：

```
//----------------------
//    ch12_8.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class Tdate{
public:
  Tdate(int m = 4, int d = 15, int y = 1995){
    month = m; day = d; year = y;
    cout << month <<"/"<< day <<"/"<< year << endl;
  }
  //其他公共成员
private:
  int month, day, year;
}; //--------------------
int main(){
  Tdate aday;
  Tdate bday(2);
  Tdate cday(3,12);
  Tdate dday(1,2,1998);
}//---------------------
```

运行结果为：

```
4/15/1995
2/15/1995
3/12/1995
1/2/1998
```

创建对象时，可以默认年，默认日与年，也可以月、日、年都默认，其他条件不允许默认。决定使用哪一个构造函数的规则和调用其他重载函数的规则相同。重载构造函数若与参数



默认值的构造函数发生冲突，则创建对象的语句会导致编译错误。参数默认与函数重载的内容可以回顾 5.8 节和 5.9 节。

例如，下面的代码在类中重载构造函数，其定义存在二义性问题：

```
class Tdate
{
  public:
    Tdate(int d)
    {
      month = 4;
      day = d;
      year = 1998;
    }
    Tdate(int m, int d = 12)
    {
      month = m;
      day = d;
      year = 1997;
    }
    //其他公共成员
  protected:
    int month;
    int day;
    int year;
    };

int main()
{
Tdate aday(11);   //error:匹配哪个构造函数?
}
```

12.7 默认构造函数

(1) **C++规定，每个类必须有一个构造函数，如果没有构造函数，就不能创建任何对象。**

(2) **若未提供一个类的构造函数(一个都未提供)，则 C++提供一个默认的构造函数，该默认构造函数是个无参构造函数，它仅负责创建对象空间，而不做任何初始化工作。**

(3) **只要一个类定义了一个构造函数(不一定是无参构造函数)，C++就不再提供默认的构造函数。也就是说，如果为类定义了一个带参数的构造函数，还想要无参构造函数，则必须自己定义。**

(4) **与变量定义类似，在用默认构造函数创建对象时，如果创建的是全局对象或静态对象，则对象的位模式全为 0，否则，对象值是随机的。**

例如，下面的代码在创建对象时，将自动调用系统提供的默认构造函数：

```
class Student
{
  //无构造函数
  protected:
    char name[20];
};
```




```
int main()
{
  Student noName; //ok: noName 的内容为随机值
}
```

上例中的类定义等价于下面的类定义：

```
class Student
{
  public:
    Student(){}  //一个空的无参构造函数
  protected:
    char name[20];
};
```

又如，下面的代码定义了一个带参数的构造函数，面对创建无参对象，将不能正确地编译：

```
#include <cstring>
using namespace std;
class Student
{
  public:
    Student(char * pName)
    {
      strncpy(name,pName,Sizeof(name));
      hame[sizeof(name) - 1] = '10';
    }
  protected:
    char name[20];
};

int main()
{
  Student noName;  //error: 无匹配的构造函数
}
```

如果增加一个无参的构造函数，就可解决这个问题：

```
//---------------------
//    ch12_9.cpp
//---------------------
#include <cstring> //用到 strncpy()
using namespace std;
//---------------------
class Student{
public:
  Student(char * pName){
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
  }
  Student(){}               //无参构造函数
private:
  char name[20];
}; //-------------------
```

```
int main(){
  Student noName;        //ok
  Student ss("Jenny");   //ok
}//---------------------
```

→ 说明的含糊性

先前的例子中创建 Tdate 类对象的方法是：

```
Tdate aday;             //为什么创建无参的对象无括号?
Tdate bday(2);          //对象的参数放在括号中
Tdate cday(3,12);       //对象的参数放在括号中
Tdate dday(1,2,1998);   //对象的参数放在括号中
Tdate aday();           //为什么不能这样声明?
```

根据 C++的语法规定，这样是声明了一个名叫 aday 的普通函数，它返回 Tdate 类对象，并且没有参数。

又如下面的代码，一个是创建对象，一个是声明函数：

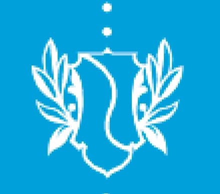
```
Tdate oneday(10);       //创建对象
Tdate oneday(int);      //声明函数
```

12.8 类成员初始化的困惑

在一个类中，如果有用户自定义的类对象作为其成员，那构造函数怎么做呢？

例如，下面的程序定义了学号类和学生类，学生类中包含学号类：

```
//---------------------
//    ch12_10.cpp
//---------------------
#include <iostream>
#include <cstring> 用到strncpy()
using namespace std;
//---------------------
int nextStudentID = 0;
//---------------------
class StudentID{       //学号类
public:
  StudentID(){
    value = ++nextStudentID;
    cout <<"Assigning student id "<< value << endl;
  }
~StudentID(){
    --nextStudentID;
    cout <<"Destructing id "<< value << endl;
  }
private:
  int value;
}; //-------------------
class Student{
public:
  Student(char * pName = "noName"){
```




```
    cout <<"Constructing student "<< pName << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
  }
private:
  char name[20];
  StudentID id;
}; //--------------------
int main(){
  Student s("Randy");
}//---------------------
```

运行结果为：

```
Assigning student id 1
Constructing student Randy
Destructing id 1
```

当学生类对象被构造时，一个学号赋予了该学生对象。

学号类 StudentID 含有保护数据成员 value，按规定不能被另一个类 Student 的类对象访问，即使 Student 类包含 StudentID 类。

Student 类包含一个成员 id，id 属于 StudentID 类，Student 构造函数不能访问 id 对象中的保护数据成员。所以，在 Student 类对象构造时，需调用 StudentID 类的构造函数来初始化对象 id。这就是类与类之间“互不干涉内政”的严格关系。

“Student s("Randy")；”语句执行步骤如下：

(1) 分配 s 对象空间，调用 Student 构造函数。

(2) 建立 s 对象空间中的结构，第一为 name[20]，第二为 id。因 id 属于 StudentID 类，于是创建过程启动了调用 StudentID 的构造函数，这时，id 的保护数据 value 得到了赋值，全局变量 nextStudentID 也得到了赋值，并且输出第一行信息。之后，返回到 Student 构造函数。

(3) 执行 Student 构造函数体。输出第二行信息，数据成员 name 得到了赋值。之后返回到主函数 main()。

如果 Student 类未定义构造函数，系统将自动地为各个数据成员(如果是类对象的话)调用 C++提供的默认构造函数。程序结束时也与此相同，系统将自动地为各个具有析构函数的数据成员调用析构函数。

我们看到在上面的例子中，Student 构造函数调用了 StudentID 的默认构造函数，但如果想调用的构造函数不是默认构造函数，那又该怎么办呢？

例如，下面的程序在创建学生对象时，赋予一个学号，希望将这个学号传给成员——学号类对象 id 保存：

```
//---------------------
//    ch12_11.cpp
//---------------------
#include <iostream>
#include <cstring>              //用到 strncpy()
using namespace std;
//---------------------
class StudentID{
```

```
public:
  StudentID(int id = 0){
    value = id;
    cout <<"Assigning student id "<< value << endl;
  }
~StudentID(){
    cout <<"Destructing id "<< value << endl;
  }
private:
  int value;
}; //--------------------
class Student{
public:
  Student(char * pName = "noName", int ssID = 0){
    cout <<"Constructing student "<< pName << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
    StudentID id(ssID); //希望将学号传给学号类对象
  }
private:
  char name[20];
  StudentID id;
}; //--------------------
int main(){
  Student s("Randy",9818);
}//---------------------
```

运行结果为：

```
Assigning student id 0
Constructing student Randy
Assigning student id 9818
Destructing id 9818
Destructing id 0
```

StudentID 的构造函数改为从参数中接受一个学号值。在 Student 构造函数内部增加了一条语句"StudentID id(ssID);"，希望将 ssID 参数值传给数据成员 id。但实际上，在 Student 构造函数中构造了一个名为 id 的 StudentID 局部对象，由于是局部对象，所以，当 Student 构造函数返回时，便析构了这个对象。从运行结果的第三、第四行中可以看到这一点。而真正希望被传递的 Randy 对象学号值却为 0，未被赋以 9818。

那么，能否像下面这样在 Student 类中直接给 id 对象初始化呢？

```
class Student
{
  public:
    Student(char * pName = "noName", int ssID);
  protected:
    char name[20];
    StudentID id(9818);   //error: 类定义是不分配空间和初始化的
};
```

"StudentID id(9818);"是创建对象语句，而不是类定义中允许的声明数据成员形式。"StudentID id=9818;"或者"StudentID id(ssID);"也都是不允许的。前者是"StudentID id(9818);"，后者 ssID 成了参数，类定义是不会调用构造函数的。因为**类是一个抽象的概念，**



并不是一个实体,并不含有属性值,而只有对象才占有一定的空间,含有明确的属性值。我们只能按照格式"类型 标识符;"去声明类的数据成员。

12.9 构造类成员

1. 类成员初始化形式

我们需要一个机制来表示"构造已分配了空间的对象成员,而不是创建一个新对象成员"。该机制应在建立对象空间的结构时反映出来,即需要出现在函数调用刚刚转入之时,函数体执行(开括号)之前。为了做到这一点,C++定义了一个新构造方式。

例如,下面的程序修改了 ch12_11.cpp 中 Student 构造函数的错误:

```
//---------------------
//    ch12_12.cpp
//---------------------
#include <iostream>
#include <cstring>
using namespace std;
//---------------------
class StudentID{
public:
  StudentID(int id = 0){
    value = id;
    cout <<"Assigning student id "<< value << endl;
  }
 ~StudentID(){
    cout <<"Destructing id "<< value << endl;
  }
private:
  int value;
}; //-------------------
class Student{
public:
  Student(char * pName = "no name", int ssID = 0) : id(ssID){  //id(ssID)为初始化类成员形式
    cout <<"Constructing student "<< pName << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\-';
  }
private:
  char name[20];
  StudentID id;
}; //-------------------
int main(){
  Student s("Randy",9818);
  Student t("Jenny");
}//-------------------
```

运行结果为:

```
Assigning student id 9818
Constructing student Randy
Assigning student id 0
```

```
Constructing student Jenny
Destructing id 0
Destructing id 9818
```

在 Student 构造函数头的后面,冒号表示后面要对类的数据成员的构造函数进行调用。ssID 是 Student 构造函数的形参,id(ssID)表示调用以 ssID 为实参的 StudentID 构造函数。

上例中,Student 构造函数头冒号后面如果是 id()的形式,表示调用 StudentID 的默认构造函数,并且可以省略。即:

```
Student(char * pName = "no name"):id()
{
  cout <<"Constructing student " << pName << endl;
  strncpy(name,pName);
  name[sizeof(name) - 1] = '\ - ';
}
```

可以写成:

```
Student(char * pName = "no name")
{
  cout <<"Constructing student " << pName << endl;
  strncpy(name,pName);
  name[sizeof(name) - 1] = '\ - ';
}
```

调用哪个构造函数完全是看参数匹配的情况,如果在 ch12_12.cpp 的 StudentID 构造函数中,参数没有默认值,即原型为:

```
StudentID(int id);
```

"Student(...)"后面无冒号形式,表明 StudentID 无默认构造函数,而 Student 构造函数的原型为:

```
Student(char * pName = "no Name", int ssID);
```

表明将调用 StudentID 的默认构造函数,于是,主函数中的两条创建 Student 对象的语句都要遇到不能匹配 StudentID 构造函数的编译错误。

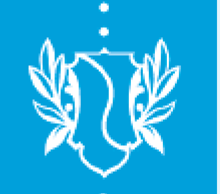
2. 常成员与引用成员初始化

冒号语法使得常量数据成员和引用数据成员的初始化成为可能。

例如,下面的程序给一个常量数据成员和一个引用数据成员初始化:

```
class SillyClass
{
  public:
    SillyClass(int& i):ten(10),refI(i){}
  protected:
    const int ten;          //常量数据成员
    int& refI;              //引用数据成员
};

int main()
{
  int i;
  SillyClass sc(i);
}
```



因为常量是不能再被赋值的,一旦初始化后,其值就永不改变。另外,引用变量也是不可重新指派的,初始化之后,其值就固定不变了。所以,不允许像下面这样对常量和引用变量赋值或重新指派:

```
class SillyClass
{
  public:
    SillyClass()
    {
      ten = 10;      //error: 常量不能赋值
      refI = i;      //error: 引用不能重新指派
    }
  protected:
    const int ten;
    int& refI;
};
```

在 SillyClass 类的构造函数进入之后(开始执行大括号后的函数体语句时),对象结构已经建立,数据成员 ten 和 refI 已经存在,所以再在构造函数体内对常量赋值或对引用变量指派就不是初始化了。常量和引用变量的初始化必须放在构造函数正在建立数据成员结构空间的时候,也就是放在构造函数的冒号后面。

对于类的数据成员是一般变量的情况,则放在冒号后面与放在函数体中初始化都一样。例如,下面两种初始化一个整数变量数据成员的方法都可以用:

```
class SillyClass1
{
  public:
    SillyClass1()
    {
      d = 10;                 //方法 1
    }
  protected:
    int d;
};

class SillyClass2
{
  public:
    SillyClass2():d(10){}     //方法 2
  protected:
    int d;
};
```

→ 说明一个变量并初始化有两种形式:

```
int main()
{
  int m = 10;               //ok
  int n(20);                //C++方式
  Student s = "Jenny";      //ok
  Student t("Danny");       //ok: 类的形式不受限制
}
```

赋值时只有一种方法：

```
m = 10;
n(20);    //此不是赋值,而是调用名叫 n 的函数
```

构造函数的冒号后面的类成员初始化不允许用第一种形式：

```
class SillyClass
{
  public:
     SillyClass():d = 10 //error
     {
     }
  protected:
     int d;
};
```

12.10　构造对象的顺序

在一个大程序中，各种作用域的对象很多，有些对象包含在别的对象里面，有些对象早在主函数开始运行之前就已经建立。**创建对象的唯一途径是调用构造函数**。构造函数是一段程序，所以构造对象的先后顺序不同，直接影响程序执行的先后顺序，导致不同的运行结果。C++给构造对象的顺序作了专门的规定。

1. 局部和静态对象，以声明的顺序构造

局部和静态对象是指块作用域和文件作用域中的对象。它们声明的顺序与它们在程序中出现的顺序是一致的。

例如，下面的程序用了 goto 语句，企图绕过变量定义：

```
//---------------------
//     ch12_13.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  int m = 5;
  if(m == 5)
    goto abc;

  int n = 0;
abc:
  cout << "m = " << m << ",n = " << n << endl; //n 已有定义
}//--------------------
```

在 C 中，其运行结果为：

```
m = 5,n = 0
```

程序中，“int n=0;”被语句“goto abc;”跳过，这将导致 C++编译器报错。即使在 C 中



可以接受,但也不是根据运行顺序来决定变量定义的顺序,而是所有的变量和对象都在函数开始执行时统一定义。统一定义的顺序正是这些变量和对象在函数中出现的顺序。

2. 静态对象只被构造一次

静态对象和静态变量一样,文件作用域中的静态对象在主函数开始运行前全部构造完毕。块作用域中的静态对象,则在首次进入定义该静态对象的函数时,进行构造。

例如,下面的程序在一个函数中定义了一个静态对象:

```
//---------------------
//    ch12_14.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class SmallOne{
public:
  SmallOne(int sma){
    cout <<"Smallone constructing with a value of"
         << sma << endl;
  }
}; //--------------------
void fn(int n){
  static SmallOne sm(n);
  cout <<"In function fn with n =  "<< n << endl;
}//--------------------
int main(){
  fn(10);
  fn(20);
}//--------------------
```

运行结果为:

```
Smallone constructing with a value of 10
In function fn with n = 10
In function fn with n = 20
```

3. 所有全局对象都在主函数 main()之前被构造

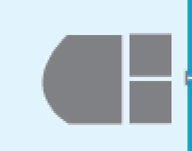
和全局变量一样,所有全局对象在主函数开始运行之前,全部已被构造。

这会给调试带来问题。当要开始调试时,所有全局对象的构造函数都已被执行,如果它们中的一个有致命错误,那么你可能永远也得不到控制权。这种情况下,该程序在它开始执行之前就死机了。

有两种方法可以解决这个问题:一是将全局对象先作为局部对象来调试;二是在所有怀疑有错的构造函数的开头增加输出语句,这样在程序开始调试时,可以看到来自这些对象的输出信息。

4. 全局对象构造时无特殊顺序

全局对象不像局部对象的构造顺序那么简单,全局对象是没有这样的控制流用来指明



其顺序的。对于简单应用的单文件程序来说，全局对象可以按照它们出现的顺序依次进行构造。但是，单文件程序只是出现在实验室或教室里，真正有用的程序都是由多个文件组成的，这些文件被分别编译、连接。因为编译器不能控制文件的连接顺序，所以它不能决定不同文件中全局对象之间的构造顺序。

例如，下面的代码是个多文件程序结构，创建了两个全局对象：

```
//student.h

class Student
{
  public:
    Student(int d):id(d){}
  protected:
    const int id;
};

class Tutor
{
  public:
    Tutor(Student& s)
    {
      id = s.id;
    }
  protected:
    int id;
};

//file1.cpp

Student ra(1234);    //建立一个 Student 全局对象

//file2.cpp

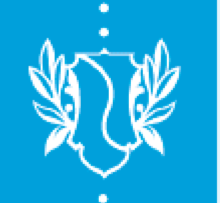
Tutor je(ra);        //建立一个 Tutor 全局对象,以使辅导员能帮助学生
```

程序中，Tutor 全局对象使用了 Student 全局对象，它隐含假定了学生类对象先于辅导员类对象构造。但是这样的设计会引起运行中不可预料的错误。为避免这个问题，不要允许一个全局对象访问另一个全局对象。

5. 成员以其在类中声明的顺序构造

例如，下面的代码在构造函数头的冒号后初始化两个成员，但是结果却并不如意：

```
//----------------------
//    ch12_15.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class A{
public:
  A(int j) : age(j),num(age+1){
```




```
        cout <<"age:"<< age <<",num:"<< num << endl;
    }
private:
    int num, age;
}; //--------------------
int main(){
    A sa(15);
}//--------------------
```

运行结果为：

```
age:15,num:2
```

由于按成员在类定义中的声明顺序进行构造，而不是按构造函数说明中冒号后面的顺序，所以 num 成员被赋的是一个随机值，并不是想赋的 16，因为这时候，成员 age 还没有被赋值，age 的内存空间中是一个随机值（此次运行，其值为 0）。

小结

构造函数是一种用于创建对象的特殊成员函数，人们调用一个构造函数来为类对象分配空间，给它的数据成员赋初值，以及进行其他请求资源的工作。每个类对象都必须在构造函数中诞生，一个类可能拥有一个或多个构造函数，编译程序为了决定调用哪个构造函数，要把对象声明中使用的变元（实参）和构造函数的参数进行比较，该过程与普通重载函数匹配函数调用的方法相同。

在包含对象成员的类对象被创建时，需要对象成员的创建，相应地调用对象成员的构造函数。然而，构造对象成员的顺序要看类中声明的顺序，而不是看构造函数说明中冒号后面类成员初始化的顺序。

构造函数尚有一些内容将在后面的章节中介绍。如何在堆中创建对象以及拷贝构造函数在第 14 章介绍，派生类的构造函数在第 16 章介绍。

练习

12.1　写出下列程序的输出。

```
#include <iostream>
using namespace std;
class MyClass
{
  public:
    MyClass();
    MyClass(int);
    ~MyClass();
    void Display();
  Protected:
    int number;
};

MyClass::MyClass()
```

```
{
  cout <<"Constructing normally\n";
}

MyClass::MyClass(int m)
{
  number = m;
  cout <<"Constructing with a number: " << number << endl;
}

void MyClass::Display()
{
  cout <<"Display a number: " << number << endl;
}

MyClass::~MyClass()
{
  cout <<"Destructing\n";
}

int main()
{
  MyClass obj1;
  MyClass obj2(20);

  obj1.Display();
  obj2.Display();
}
```

12.2 创建一个 Employee 类，该类中有字符数组，表示姓名、街道地址、市、省和邮政编码。把表示构造函数、ChangeName()、Display()的函数原型放在类定义中，构造函数初始化每个成员，Display()函数把完整的对象数据打印出来。其中的数据成员是保护的，函数是公共的。

12.3 修改练习 12.2 中的类，将姓名构成类 Name，其名和姓在该类中为保护数据成员，其构造函数为接收一个指向完整姓名字符串的指针，其 Display()函数输出姓名。然后将 Employee 类中的姓名成员（字符数组）换成 Name 类对象。

将所有原型化的函数加上成员函数定义，作为类的内部实现文件。

构成完整的类库定义，要求类定义与类的成员函数定义分开。

12.4 根据练习 12.3，编制主函数如下，构成一个完整的多文件程序。

```
int main()
{
  Employee em("Mark Brooks","5 West St.","Revere","CA","12290");

  char buffer[255];
  em.Display()

  em.ChangeName("Richard Voss");
  em.Display();
}
```

第13章　面向对象程序设计

C++区别于 C 的特征是 C++支持面向对象程序设计。在知道了 C++中如何创建类后，必须搞清什么是面向对象程序设计，类适用于现实世界中的哪些问题，才能真正进行面向对象的思考和编程。学习本章后，应该了解面向对象编程方法与结构化编程方法的区别；学会抽象和分类以及简单的面向对象程序设计。

13.1　抽象

面向对象程序设计基于两个原则：抽象和分类。

抽象与具体相对应。一个人名是抽象，它代表某人的一切属性，包括身高、体重、文化程度等。抽象是对具体事物的描述的一个概括。

试想一下用微波炉炖鸡蛋。在碗里打两个鸡蛋，放上一点调料，把它整个放进微波炉里，烘烤 5 分钟。

使用微波炉的步骤是，先打开门，把制作的原料放进去，然后关好门并按微波炉前面控制板上的有关按钮，它就开始工作了。

使用微波炉，人们处于下面的状态：

(1) 不用重新设计布局，不用改变微波炉的内部结构即可使它工作。人们使用微波炉，只需跟微波炉的面板打交道。微波炉有一个接口，就是微波炉的面板，板上有所有的控制按钮和时间显示。微波炉的所有功能都是通过面板控制获得的。

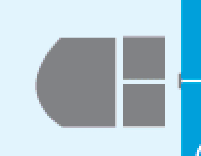
(2) 不用重新编制软件来驱动和控制微波炉中的微处理器，即与上次使用微波炉的目的无关。

(3) 不用了解微波炉的内部结构。

(4) 一个微波炉的设计师，知道微波炉的内部一切设计细节，但在生活中微波炉只是用于烧菜热菜，而无须考虑其工作原理。

现实生活中，为了减少必须处理的事情，我们是在某一程度的细节中生活的。在面向对象的计算机世界中，这种细节程度就叫抽象。

在做菜时，人们仅仅把微波炉看成是一个厨房用品在使用，不会考虑微波炉的内部结



构。既然只是通过它的接口来使用微波炉，按照显示的提示去做，就不会使微波炉进入不正常的工作状态而损坏微波炉或把菜烤焦。

如果正常操作下，却被炉壁烫伤了手，或微波炉冒出了火花等，那就是微波炉的质量问题。如果误操作引起菜烧焦了，或烧不熟，那就要调整操作。这在面向对象程序设计中是分工明确的两种编程：一种是面向对象应用程序设计；一种是类库设计。它们都属于面向对象程序设计范畴。

如果操作微波炉之前改动了微波炉的内部结构，或更换了一些电路，那么任何烫伤等事故概由操作人负责。面向对象程序设计中，这就像是修改了类库。那么，类库的维护也应由该程序员负责到底。

用面向对象的方法描述在微波炉中炖蛋的过程时，首先定义这个问题中对象的类型：蛋，微波炉，还有调料；然后，着手设计制作这些对象的模型，即考虑微波炉的制作，鸡蛋的采购等。

当做“制作微波炉”这项工作时，程序设计在具体的对象一级上，这时候，不用考虑鸡蛋如何做。

当微波炉做成之后，就可以进入下一个抽象级，开始考虑炖鸡蛋的调制过程。这时候，不用考虑微波炉的制作，而可直接在微波炉上进行操作。

操作程序可像下面这样：打碎两个蛋，放点水和调料等，在微波炉中烧 5 分钟。这就是更高级抽象的描述，也是面向对象程序设计中主程序的描述。这样的描述既简单明了又完整，但这不是一个结构化程序的描述。

结构化程序是使微波炉的外壳和内部结构与鸡蛋、调料、水同处于一个程序环境中，其程序显示出层层的函数调用结构。控制从手到面板，从面板进入微波炉内部，在复杂的内部电路的逻辑中流动，最后发出“来取吧”的声音。在这个环境中，很难理解抽象及其程度。程序中没有对象，没有能隐藏事物固有复杂性的抽象。制作炖蛋是微波炉制作的目标之一，所以必须首先是制作微波炉的专家，等到下次制作红烧鱼时，又要重新制作一个专门烧鱼的微波炉了。

13.2 分类

层层分类，使概念逐渐细化，即具体化。相反，归类的结果，便是逐步抽象的过程。

例如，我问：什么是桑塔纳？一般的回答是：它是一种小轿车。如果我又问：什么是小轿车？一般的回答是：它是一种小汽车。如果我再问：什么是小汽车？一般的回答是：它是一种汽车。那么我又问：什么是汽车呢？一般的回答是：汽车是一种交通工具，等等。

因为人们理解的桑塔纳是我们生活中称为小轿车一类东西的一个实例。也就是说，桑塔纳是一种特殊类型的小轿车，而小轿车是一种特殊的小汽车。事实上，桑塔纳还不是具体的实物，它只不过是一个名字，代表所有的桑塔纳牌小轿车。

在面向对象的计算机世界中，我们把一辆实实在在的桑塔纳小轿车称作类(class)桑塔纳的一个实例(instance)或者说是对象(object)。类桑塔纳是类小轿车的一个子类，而类小轿车又是类小汽车的一个子类，类小汽车是类汽车的一个子类，类汽车又是类交通工具的子类，等等。

在我们生活的这个世界上，每个事物、每件东西都按规则分了类。做这项工作，是为了



减少我们必须记住的东西个数。

例如,广告上说,某某牙膏具有特殊的止血功能。我们便知道,该产品是一种牙膏,具有牙膏的一切属性,外壳形状、颜色、膏状、清洁牙齿的功能、用在牙刷上等。除此之外,它还具有止血的特殊疗效。所以,牙膏是抽象的概念,它使我们理解一切具有牙膏属性的东西。任何牙膏的新产品,只要描述一下除了牙膏公共属性之外的特殊属性即可。

桑塔纳小轿车的外形与其他小轿车不一样,内部的若干特性也与其他小轿车不一样,除此之外,桑塔纳小轿车便是一般意义上的小轿车了——有4个轮子,有方向盘,等等。

在结构化程序设计方法中没有分类的概念。因为结构化程序设计强调的是过程的功能划分,注重功能性。每一个功能,都靠自己解决。如果一个功能与另一个已存在的功能实现类似,也不能从中为我所用。

在面向对象的程序设计中,对象被分成类。类又是层层分解的,这些类与子类的关系可以被规格化地描述。描述了类,再描述其子类,就可以只描述其增加的部分。所有子类层次上的编程,都只需在已有的类的基础上进行。

分类是面向对象程序设计的需要。在这之前,在结构化程序设计中存在一些问题:

(1) 在结构化程序中,总是将制作汽车和驾驶汽车混在一起。所以,要到达目的地,必须同时具有汽车制造技术和汽车驾驶技术。这使得编程难度大大增加。驾驶汽车是简单的,在现有某种型号汽车的基础上,制造新型的汽车也是简单的。只要分离汽车制造和汽车驾驶,就能使程序编制简单化。为了分离汽车制造与汽车驾驶,就要将汽车从过程中分离出来,成为一个独立的概念,并用分类技术使汽车的描述与实现简单化。只要说明与现有的某种汽车的不同之处,即可完整地描述所使用的汽车。

(2) 在结构化程序中,由于汽车驾驶与汽车制造是交错实现的,所以要更换汽车很难。这使得程序不灵活。我们说,要到达目的地,不一定非得要某一型号的汽车,只要满足一定的装载量和速度等要求,其他的汽车完全可以代替。一旦有了汽车的分类,就有关于汽车的操作描述,根据该描述,完全可以从汽车的分类中找到其他同样功能的汽车。

(3) 在结构化程序中,如果改变了到达的目的地或运载量,就要将程序推翻重来。这意味着要重新制造汽车。有了分类的概念,就可以在已存在的汽车分类(重用)中,轻易地定义出满足要求的汽车,而不必修改运载的过程。

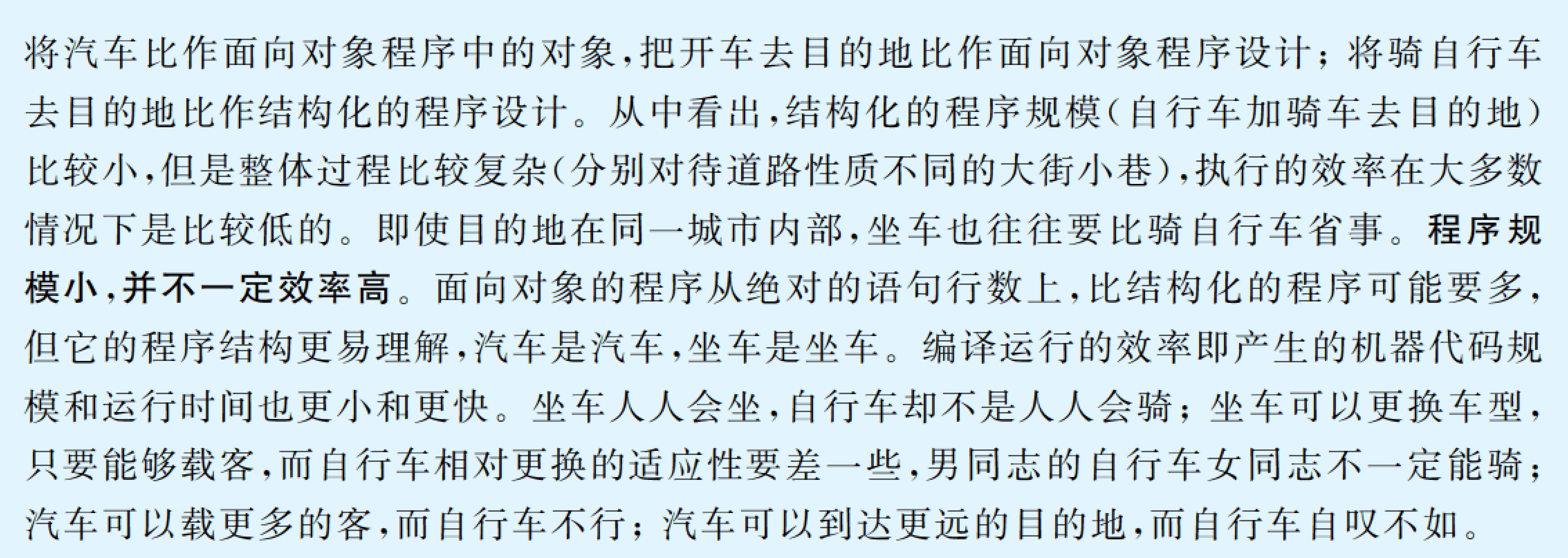
分类是理解抽象的重要手段,也是面向对象程序设计中的重要概念。掌握了分类方法,就能理解面向对象程序设计的过程。

13.3 设计和效率

有人说,面向对象程序设计,相比结构化程序设计,不能产生出高效的程序。

思考一下我们所使用的交通工具。试想一个体力很好的人(比作熟练的程序员),骑着自行车,穿行在大街小巷中,轻松自在地到达目的地。他走的路,是捷径。可是,如果目的地是较远的地方,例如是另一个城市,那么骑车去,还能轻松自在吗?也许他会选择坐汽车,甚至让其他驾驶员带他到目的地。这样,他就无须辨认任何路名路标,还可沿途欣赏田园风光。坐汽车比骑自行车更方便了。然而,有人会说,汽车比自行车要庞大,耗费更多的费用。

可以将高速公路比作面向对象程序设计方法,把街道马路比作结构化程序设计方法;


将汽车比作面向对象程序中的对象，把开车去目的地比作面向对象程序设计；将骑自行车去目的地比作结构化的程序设计。从中看出，结构化的程序规模（自行车加骑车去目的地）比较小，但是整体过程比较复杂（分别对待道路性质不同的大街小巷），执行的效率在大多数情况下是比较低的。即使目的地在同一城市内部，坐车也往往要比骑自行车省事。**程序规模小，并不一定效率高**。面向对象的程序从绝对的语句行数上，比结构化的程序可能要多，但它的程序结构更易理解，汽车是汽车，坐车是坐车。编译运行的效率即产生的机器代码规模和运行时间也更小和更快。坐车人人会坐，自行车却不是人人会骑；坐车可以更换车型，只要能够载客，而自行车相对更换的适应性要差一些，男同志的自行车女同志不一定能骑；汽车可以载更多的客，而自行车不行；汽车可以到达更远的目的地，而自行车自叹不如。

汽车必须在大街或高速公路上开，小巷中不能开。划分出类，就成了面向对象程序设计。自行车既可在大街也可在小巷中骑，小程序也可采用面向对象程序设计方法。采用面向对象的程序设计方法才有高效运行的效果。汽车在高速公路上行驶比之自行车快很多。唯一“不足”的是，汽车比自行车要耗费更大的成本。这也是很自然的，因为面向对象程序设计的基础是以大量类库作为产品被生产出来。各种汽车就是大量供使用的产品，没有汽车，有了高速公路也白搭。汽车用来为大众所用（重用）。随着工业化程度的提高，人们感受到的是方便，人们也适应了鳞次栉比、纵横交错的交通网。人们会逐渐忘却结构化程序设计，而自然采用面向对象程序设计。

一些小程序，可以通过过程化的程序设计技巧和优化，小幅度提高运行速度，但往往以牺牲可读性为代价，给维护造成大量的困难。一旦程序规模扩大，程序的可读性和可维护性，甚至连结构化的程序设计都感到力不从心。

在现实生活中，能解决问题的小规模程序是很少的。所以，可以说，面向对象程序设计比结构化程序能够产生出更加有效的程序。而且，面向对象的程序，其可读性、可维护性都比结构化程序好。

→ 何为效率

编程效率分为程序设计效率和程序运行效率。

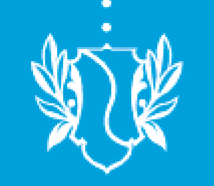
设计效率不能只看其在简单结构与规模下的设计方法，还要看其在复杂结构和规模下的设计方法，要看其方法是否能够适应大型复杂设计的能力，并最终节省大量的资源。

运行效率不能只看其运行速度的快慢，也不能只看其占据的存储空间，要综合地去比较时间和空间，才能客观地评价程序运行的效率。

面向对象程序设计通过分层的方法有效拆解程序元件——对象，抽象化程序模块进而有效地封装代码，成功实施类编程与应用编程的分工合作来增强代码安全性，通过有效的代码重用来增强代码维护的方便性。其优势主要体现在高效的编程设计上，其亦必然带来合理调配资源之高效运行的程序，否则其设计方法本身就不可取。

13.4 讨论 Josephus 问题

Josephus 问题，我们前面曾经讨论过，如果起点可以是任意一个小孩，那么在解决 Josephus 问题中，还要另给出起点位置。



根据前面讨论的内容,可以得到一个处理过程的描述:

```
//Josephus 问题解答

建立小孩结构类型
初始化小孩数,开始位置,数小孩个数
分配小孩结构数组
for 初始化结构数组(构成环链)
   挂接下一个数组元素
   小孩编号
   输出编号
endfor

转到开始位置

while(小孩数多于一个)
   数小孩个数(一个循环)
   出列小孩
   将该小孩从环链中删除
endwhile

输出得胜者

返还结构数组空间
```

该程序是一个过程化的实现。在前面章节给出的解答中,都是按这种思路实现的。相对来说,它要求更高的程序设计技巧。该问题的解决紧紧依赖于所定义的数据结构,程序员必须对 Josephus 问题的解决办法非常清楚,并且对实现其解答的数据结构操作也十分内行。编程是高度专业化的,我们在前面看到的解答就是技巧性相当高的两种实现。

但是,不能对所有的问题都用这样的直观分析法。当问题趋向复杂化时,就要将问题分割成一个个小块,从而最终解决问题。

结构化程序设计方法按功能分割问题。面向对象程序设计按对象分割问题。

13.5 结构化方法

上节描述的程序流程是比较清楚的。但是我们仍认为它太复杂,对它进行功能分解,也即对下面各项设计出各个功能子块:

(1) 初始化小孩数,开始位置,数小孩数;

(2) 初始化环链表(采用链表数据结构来解);

(3) 数小孩;

(4) 处理未获胜的小孩。

分解之后,主程序描述变得短小了:

```
//Josephus 问题解答

建立结构
初始化小孩数,开始位置,数小孩个数
初始化环链表(采用链表数据结构来解)
```

```
转到开始位置(一个循环)

处理未获胜的小孩

输出得胜者
返还结构数组空间
```

其中，初始化小孩数，开始位置，数小孩个数描述为：

```
键入小孩数、开始位置、数小孩个数
小孩数校验
开始位置校验
数小孩个数校验
```

初始化环链表描述为：

```
分配结构数组

for 初始化结构数组(构成环链)
  挂接下一个数组元素
  小孩编号赋值
  输出小孩编号
endfor

返回环链表
```

处理未获胜的小孩描述为：

```
while(小孩数多于一个)
  数小孩个数(一个循环)
  出列小孩
  将该小孩从环链中删除
endwhile
```

数小孩描述为：

```
for(从 1 到数小孩间隔数)
  开始位置挪到下一个小孩
endfor
```

主程序描述中，只涉及一个个功能调用，没有循环，比之前面将整个问题放在一起解决要简单和容易理解一些。每个功能调用相对一个大程序来说，也简单一些，这样便容易把握。这对程序设计的可读性来说，进了一大步，其他程序员很容易理解此程序，并且乐意对其进行维护。

主程序中省略了细节，而在真正实现时，再进一步具体化。这正是更为抽象的程序设计(面向对象程序设计)之技术与基础。

但从整个问题解决来说，它又显得复杂，用户只关心给出小孩数、开始位置和每隔多少小孩出列一个小孩的间隔以及需要得到获胜者的位置。除此之外，任何内部实现的细节，都是多余的。如果把这些内部实现的细节分离出来，让更专业的人员来实现，岂不更符合社会化大生产的要求吗？

然而，结构化程序做不到，只能由程序员来设计解答整个过程。可以将其中的一部分作为函数调用分给别人实现，但他自己的专业化程度甚至更高。即他必须全盘把握程序中的



数据结构(本例中的小孩结构和链表),并在频繁调用其他函数的过程中,为维护这些数据结构和数据操碎了心,且"吃力不讨好",由于数据结构和数据对所有函数都可见,很难把握数据的修改来自哪个函数,这些数据的安全性得不到保障。难道软件开发真的只能由计算机专家来完成吗?

计算机发展到今天,有许多软件产品都是现成的,它们以类库的形式提供,只要拿来用就可以了。

一个电视机电路设计专家,其家里摆放的电视机与别人从市场上买到的一样,能够欣赏到的节目与别人也一样。使用电视机真的无须电视机设计技术。

Josephus 问题的结构化分析方法,是将该问题按功能分解,然后按必要的顺序实现之。功能的划分并不严格,可按复杂程度分,对有些简单的功能不必划分出去,如输出小孩编号、小孩脱离链表等。

13.6 结构化方法的实现

现在,我们不得不面对这样的结构化程序,这个程序的组织,是比较紧凑和高效的,但是,程序员除了懂 Josephus 算法之外,对于数据和链表的维护,也用尽了程序技巧,这在面向对象程序设计看来,根本没有必要:

```
//----------------------
// Josephus 问题解法三
//   jose3.cpp
//----------------------
#include<iostream>
#include<iomanip>
#include<cstdlib>              //用到 exit(1)
using namespace std;
//----------------------
struct Jose{                   //小孩结构
  int code;                    //存放小孩编号
  Jose* next;                  //用于指向下一个小孩结点
}; //--------------------
int n;                         //小孩数
int begin;                     //开始位置
int m;                         //数小孩间隔
Jose* pivot;                   //链表哨兵
Jose* pCur;                    //当前结点指针
//----------------------
void assign();                 //赋初值,返回 1:成功,0:失败
Jose* initial();               //初始化环链表
void count(int m);             //数 m 个小孩
void process();                //处理所有未获胜小孩
//----------------------
int main(){
  assign();
  Jose* pJose=initial();       //初始化结构数组
  process();                   //处理所有未获胜小孩
  cout<<"\nthe winner is "<<pCur->code<<endl;
```

```
    delete[] pJose;                    //返还结构数组
  }//--------------------
  void assign(){                       //赋初值
    cout <<"please input the number,begin,count:\n";
    cin >> n >> begin >> m;
    if(n < 2){                         //小孩数校验
      cerr <<"bad number of boys\n";
      exit(1);
    }
    if(begin < 0){                     //开始位置校验
      cerr <<"bad begin position.\n";
      exit(1);
    }
    if(m < 1 || m > n){                //数小孩个数校验
      cerr <<"bad interval number.\n";
      exit(1);
    }
  }//--------------------
  Jose * initial(){                    //链表初始化
    Jose * px = new Jose[n];
    for(int i = 1; i <= n; i++){
      px[i - 1].next = &px[i % n];
      px[i - 1].code = i;
      cout << setw(4)<< i;
    }
    cout << endl;
    pCur = &px[(n + begin - 1) % n];   //指向结构数组开数第一个元素
    return px;
  }//--------------------
  void count(int m){                   //数 m 个小孩
    for(int i = 0; i < m; i++){
      pivot = pCur;
      pCur = pivot -> next;
    }
  }//--------------------
  void process(){                      //处理获胜者决出之前的所有 n - 1 个小孩
    for(int i = 1; i < n; i++){
      count(m);                        //数 m 个小孩
      cout << setw(4)<< pCur -> code;
      pivot -> next = pCur -> next;    //小孩脱链(前一小孩指向后一小孩)
      pCur = pivot;                    //当前指针指向前一小孩呈开数态
    }
  }//--------------------
```

上例程序从规模(语句行数)上看,超过了过程化的程序;从程序的模块结构看,可读性有一些提高。但是,几乎每个函数都要用小孩数,不止一个函数要用开始位置和数小孩个数,所以不得不设置全局变量。这些全局变量使得函数的独立性大为降低,函数本身具有的黑盒特性遭到破坏。函数调试过程中数据相互依赖,这使程序开发和维护的难度大大提高。程序的可读性也大打折扣。

主程序的实现体现了系统设计思想,但实现过程却使程序员无法摆脱数据结构的细节。从数据变量的命名、类型,到链表结构的指针、结构数组的堆分配、返还等,无不要求程序员具有高度的专业素质,他们必须懂得业务和计算机专业两方面的知识。



13.7 面向对象方法

在面向对象的分析和设计中,我们执行下面的步骤:

(1) 找出类;

(2) 描述类和类之间的联系;

(3) 用类来定义程序结构。

找出类主要靠经验,程序员可由一系列候选类开始,然后考虑哪一个是最基本的以及哪一个是第二位的或者是被引出的。候选类由以下各项可找出:

(1) 有形的、可视的或描述的东西。如电视机、微波炉、桌子、问题等;

(2) 角色。如操作电视机的人,桌子上摆放的东西,问题中涉及的链表结构,等等;

(3) 事件。如操作电视机的亮度,桌子的移动,问题中描述的操作,等等。

对于很复杂的程序,程序员必须做一个完全的分析,并充分了解问题的各项细节,然后把问题分类:哪些跟电视机有关,哪些跟桌子有关……抽象出描述的对象。

对于简单的问题,通过问题陈述和列出名词表,可以帮助解决问题。例如一个解决Josephus问题的名词表:

```
Josephue 问题
小孩
链表
开始位置
小孩数
每数若干小孩的间隔
去掉一个小孩
描述胜利者
等等
```

其中,"开始位置","去掉一个小孩","每数若干小孩"都与链表有关。链表还包括分配一个结构数组,初始化结构数组、环链等。这些名字有些是类,有些是组成类的属性,有些是描述类的行为。

作为一个要处理的问题Josephus,我们可以把它看作一个类。另外,链表包括其数据属性(结构数组,当前位置等)和链表操作(移动小孩位置,去掉小孩等),所以可以把它看作另一个类。

这时,要把类中要用到的数据属性(数据成员)和操作(成员函数)描述清楚。

一般来说,**有一个数据属性,就有该数据属性的访问操作**。每个数据成员和成员函数的取名也是一个环节,应该使用易懂的组合单词,没有用的名字应删掉。

例如,图13-1描述了Josephus问题类和环链表类的卡片。

Josephus 类	
Initial	Boynumber
Getwinner	BeginPos
	Interval

Ring(环链表)类	
Clear	First
Print	Pivot
Count	Current

图 13-1　类的卡片描述



图中的卡片左面是成员函数描述，表示类的外部接口；右面是数据成员描述。有了这些类的描述，就可以设计程序了：

```
//主函数
定义一个 Jose 类对象
赋初值 initial
求获胜者 getwinner
```

从中我们感受到这种编程并不需要涉及 Josephus 问题和链表结构的复杂细节。由于 Jose 类包含了与环链表类的通信联络，所以在主函数中丝毫不需要涉及环链表类，而且，Jose 类中的具体实现也并未涉及，只要提供该类的实现，那么编程就可以这么轻松和简单。**面向对象程序设计使用户既不需要懂计算机太多，也不需要懂业务太多。**

下面我们来看 Jose 类的实现者是如何来做这项工作的：

```
Jose 类
{
  接口(成员函数)
    构造函数
    赋初值 initial
    求获胜者
  内部数据成员
    小孩数
    开始位置
    数小孩个数
}
```

其中，作为接口的成员函数描述为：

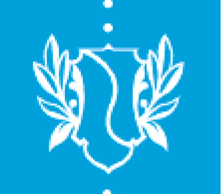
```
赋初值 initial(成员函数)
{
  键入小孩数，校验
  键入开始位置，校验
  键入数数间隔，校验
}

求获胜者(成员函数)
{
  初始化环链表

  转到开始位置
  while(小孩数多于一个)
    数 m 个小孩(一个循环)
    出列小孩
    将该小孩从环链中删除
  endwhile

  返回得胜者
}
```

描述类总是标准的，既有表示类的操作(成员函数)，也有表示类的属性(数据成员)。一个解决 Josephus 问题的实现者，完全可以将该类商品化。因为它是标准的类描述，所以，用



户拿来,按照其接口,立即可以用来解决有实际数据的 Josephus 问题。

在 Jose 类中,要用到环链表类,所以 Jose 类要包含环链表类的描述:

```
ring(环链表)类
{
   接口(成员函数)
      构造函数
      根据数数间隔数小孩
      输出当前位置的小孩
      从环链中去掉当前小孩
      析构函数
   内部数据成员
      小孩结构数组指针
      当前小孩指针
      小孩哨兵指针
}
```

其中,作为接口的成员函数描述为:

```
构造函数(成员函数)
{
   分配小孩结构数组

   初始化结构数组
      小孩编号
      输出编号
      构成环链表

   当前小孩指针位置
}
```

实现 Jose 类的程序员,也可以将环链表类拿来为自己所用。C++中,提供了标准的链表类库,只要稍加继承,就能派生为环链表类。在 20.7 节,我们有使用标准模板类库解决 Josephus 问题的解答。

在面向对象程序设计中,我们看到,程序设计主要是描述类,大大小小的类都是标准的描述结构。这使得程序结构一般化,程序的可读性大为改观。而真正的程序主体,在这里就是 Josephus 问题的解答,只有短短的 3 条语句。事实上,它只是构造了类对象,启动(调用成员函数)了一下其中的一个类的行为。完全体现了:

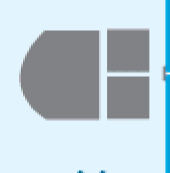
```
程序 = 对象 + 对象 + ……
```

因而,面向对象程序设计,归结为类的设计。只要将问题中的对象层次划分清楚,C++就能将它用语言描述出来。描述了类结构(类声明与类定义),余下的问题就是简单地定义对象和让对象表现自己,问题的解答也就差不多有了。这就是面向对象程序设计的基本方法。

面向对象程序设计可以将程序员分成两类:

一类是面向对象应用程序设计。他(她)们无须了解类的实现细节,就像使用微波炉那样,这要比结构化程序设计简单得多。

另一类是类库设计。他(她)们为面向对象程序设计提供“素材”。这些素材涉及各个领域,由各领域的专业人员来设计完成。他(她)们需要了解特定类的知识,如 Josephus 问题



的解答。

由于知识以类为单位,类是规范格式的,所以每个类的描述也很轻松,并使程序设计的分工合作更易于进行。因此,只要将问题归结为对象以及对象的联系,便可得到问题的解答。

这样的程序可读性是良好的,可维护性也是良好的,因为程序的结构更加规范化,它以抽象层来划分问题的解,而且问题的描述几乎就是程序的实现。

13.8 面向对象方法的实现

下面的程序是上面描述的面向对象程序设计方法的具体实现。该程序是个多文件结构,工程文件中包括 3 个源文件:

```
// *********************
// Josephus 问题解法四
// jose4.prj
// *********************
ring.cpp          //头文件 ring.h 中 Ring 类的实现
jose.cpp          //头文件 jose.h 中 Jose 类的实现
jose4.cpp         //主函数定义
// *********************
```

源文件和在源文件中包含的头文件分别为:

```
//---------------------
//    ring.h
//---------------------
#ifndef RING
#define RING
struct Boy{                          //小孩结构作为链表结点
  int code;
  Boy * next;
}; //--------------------
class Ring{                          //环链表类定义
public:
  Ring(int n, int beg);
  void Count(int m);                 //数 m 个小孩
  void PutBoy(bool f = 0)const;      //输出当前小孩的编号
  void ClearBoy();                   //将当前小孩从链表中脱钩
  void Display();                    //输出圈中所有小孩
  ~Ring();
private:
  Boy * pBegin;
  Boy * pivot;
  Boy * pCurrent;
}; //--------------------
#endif

//---------------------
//    ring.cpp
//---------------------
#include"ring.h"
#include <iostream>
```




```
#include <iomanip>
using namespace std;
//----------------------
Ring::Ring(int n, int beg){
  pBegin = new Boy[n];                    //分配小孩结构数组
  pCurrent = pBegin;
  for(int i = 1; i <= n; i++){
    pBegin[i - 1].next = pBegin + i % n;  //将结点链起来
    pBegin[i - 1].code = i;               //小孩编号
    PutBoy();
    pCurrent = pCurrent -> next;
  }
  pCurrent = &pBegin[n + beg - 2];        //当前小孩位置在最后一个编号
}//---------------------
void Ring::Count(int m){
  for(int i = 1; i <= m; i++){
    pivot = pCurrent;
    pCurrent = pCurrent -> next;
  }
}//---------------------
void Ring::PutBoy(bool f) const{
  static int numInLine;
  if(f){                                  //打完一轮则重置新一轮
    numInLine = 0;
    return ;
  }
  if(numInLine++ % 10 == 0)
    cout << endl;
  cout << setw(4)<< pCurrent -> code;
}//---------------------
void Ring::ClearBoy(){
  pivot -> next = pCurrent -> next;
}//---------------------
Ring::~Ring(){
  delete[] pBegin;
}//---------------------
void Ring::Display(){                     //从当前结点开始打印整个环
  for(Boy* pB = pCurrent; (pCurrent = pCurrent -> next)!= pB; )
    PutBoy(0);
  PutBoy(1);                              //打完一轮
}//---------------------

//----------------------
//    jose.h
//----------------------
#ifndef JOSE
#define JOSE
class Jose{
public:
  Jose(int boys = 10, int begin = 1, int m = 3){
    numOfBoys = boys;
    beginPos = begin;
    interval = m;
  }
```

```
void Initial();
    void GetWinner();
  private:
    int numOfBoys;
    int beginPos;
    int interval;
  }; //--------------------
  #endif

  //---------------------
  //     jose.cpp
  //---------------------
  #include"ring.h"                    //告诉编译,本文件中将使用Ring
  #include"jose.h"
  #include<iostream>
  #include<cstdlib>                   // 用到exit()
  using namespace std;
  //---------------------
  void Jose::Initial(){
    int num,begin,m;
    cout<<"please input the number of boys," \
          "begin position,interval per count :\n";
    cin>>num>>begin>>m;
    if(num<2){
      cerr<<"bad number of boys\n";
      exit(1);
    }
    if(begin<0){
      cerr<<"bad begin position.\n";
      exit(1);
    }
    if(m<1||m>num){
      cerr<<"bad interval number.\n";
      exit(1);
    }
    //输入数据都合法时,予以赋值
    numOfBoys=num;
    beginPos=begin;
    interval=m;
  }//--------------------
  void Jose::GetWinner(){
    Ring x(numOfBoys,beginPos);       //小孩围成圈,转到开始位置
    for(int i=1; i<numOfBoys; i++){   //处理除了获胜者之外的所有小孩
      x.Count(interval);              //数小孩
      x.PutBoy(0);                    //输出小孩编号
      x.ClearBoy();                   //当前小孩脱离环链
    }
    x.Count(1);
    cout<<"\nthe winner is ";
    x.PutBoy(0);                      //获胜者
    cout<<"\n";
  }//--------------------

  //---------------------
```

```
//    jose4.cpp
//----------------------
#include"jose.h"        //告诉编译,本文件中将使用Jose类
#include<iostream>
using namespace std;
//----------------------
int main(){
  Jose jose;            //建立Josephus问题对象
  jose.Initial();
  jose.GetWinner();
}//---------------------
```

图13-2表示了该程序中文件之间的关系。

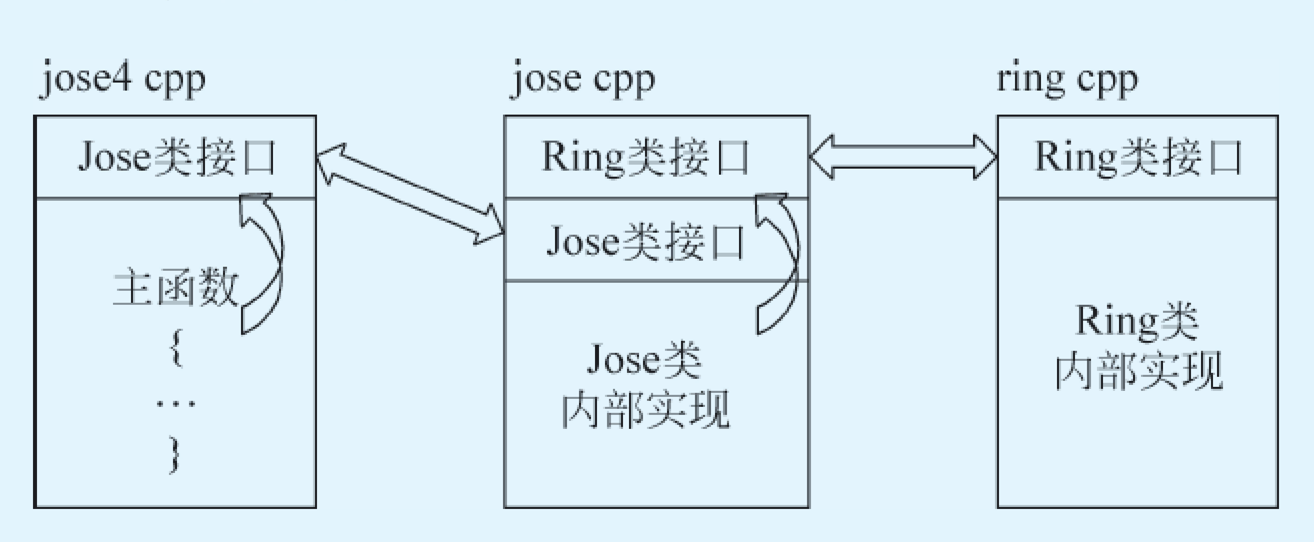


图13-2　程序中源文件的相互关系

图中,主函数要用到Jose类,Jose类要用到Ring类。

类Ring定义了环链表的所有操作与数据。在头文件ring.h中,定义了类的操作界面,因为要用到小孩结构Boy的名字,所以在该处定义了Boy结构。而在Ring类的成员函数定义文件ring.cpp中,无须再次定义Boy结构,因为该文件包含了ring.h头文件。

如果哪个程序要用ring类,只要包含ring.h头文件即可,无须涉及ring类的实现细节(即成员函数的实现代码)。ring.h头文件主要告诉用户其成员函数的功能和调用方法(函数原型)。在这里是:

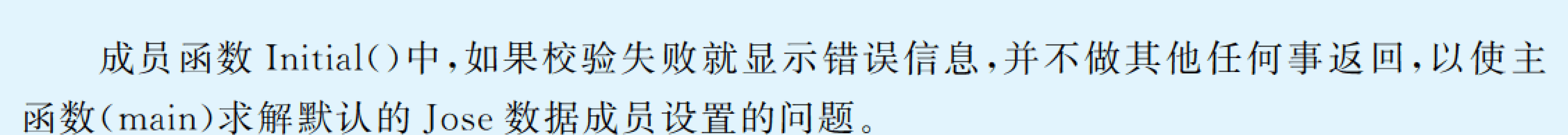
```
void Ring::Count(int m);      //数m个小孩
void Ring::PutBoy();          //输出当前小孩编号
void Ring::ClearBoy();        //将当前小孩从链表中脱钩
```

另外还包含构造函数和析构函数。

至于保护数据,它们不是面向对象程序设计中类的界面。也就是说,用户使用(重用)类时,无须类中保护数据和私有数据,因为这些数据成员都由成员函数共享使用,而不能直接由用户去访问。但它们安排在类定义中,是类机制实现的需要。所以,使用类时,我们包含类的头文件,只使用其成员函数,而无须关心保护数据和私有数据。

所有的类,都是以类定义(类的头文件)作为类的界面,这样的一致性,使得程序结构有条不紊地组织和划分,程序可读性和可维护性大大增强。

同理,Jose类也是这样组织的。Jose类中使用了Ring类,只要包含ring.h头文件即可。Jose类的头文件jose.h定义了类Jose的界面,头文件中尚无涉及Ring类,所以无须包含ring.h。在Jose类的成员函数定义(jose.cpp)中,需要用到Ring类对象,所以该文件包含了ring.h头文件。同时也包含了Jose类自己的头文件(jose.h)。



成员函数 Initial()中,如果校验失败就显示错误信息,并不做其他任何事返回,以使主函数(main)求解默认的 Jose 数据成员设置的问题。

我们看到成员函数 GetWinner(),是求解 Josephus 问题的主算法。但由于使用了类,其算法十分简单。编程成了抽象描写过程的“艺术”,只要遵循面向对象的程序设计,那么大程序的开发再也不用让主设计者一个人处于必须了解系统所有的方方面面的地位,而他完全可以在划分了类之后,抽象且艺术化地编程。

在主程序中,main()函数只包括了 3 条语句,我们再次领略了面向对象程序设计的抽象性。抽象比具体容易记忆,抽象使问题能够简单表达,抽象意味着当前无须深究。

13.9 程序维护

作为 Josephus 问题的一个扩展,可以要求有若干个获胜者。这时,便涉及程序维护问题。求一个和几个获胜者,都是求获胜者,所以对面向对象的应用程序来说,最多传递一个获胜者个数参数给 Jose 类对象的求获胜者成员函数。

我们将 Josephus 问题(类)的实例看作是具备任何求解条件的对象,这样,面向对象的应用程序仍然为建立 Jose 类对象,让其具备求解条件,求获胜者。这时程序代码不必作任何改动。

在 Jose 类中,保持外部接口不变。这是面向对象应用程序不必改动的根本原因。但在具体实现上,由于要求几个获胜者,所以要适当修改类库代码。

(1) 在 Jose 类中增加一个获胜者个数的数据成员。

(2) 在具备求解条件的 Initial()成员函数中,需要输入获胜者个数,并落实对其的校验。

(3) 在求获胜者 GetWinner()成员函数中,要修改处理小孩的循环次数。

(4) 输出获胜者的过程要扩充,并不局限于打印一个获胜者,而是要将现存小孩圈的所有小孩都作为获胜小孩打印。由此,原有的 Ring 类中的 PutBoy()成员函数要作修改。

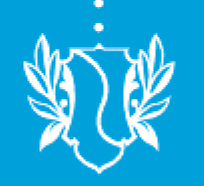
(5) 为此,Ring 类中增加 Display()成员函数,它负责将小孩圈中所有小孩都打印一遍。

(6) 由于 Ring 类的外部接口发生变化,调整 Jose 类的内部实现,即修改 GetWinner()成员函数,打印所有获胜者。

(7) Ring 类自身发生变化,调整自身,使之更趋合理。此处即修改其构造函数。

于是,对上面的程序,作适当的修改如下,成为解决 Josephus 问题的第五个解答。该工程文件为:

```
//**********************
// Josephus 问题解法五
// jose4.prj
//**********************
ringx.cpp        //头文件 ringx.h 中 Ring 类的实现
josex.cpp        //头文件 josex.h 中 Jose 类的实现
jose5.cpp        //主函数定义
//**********************
```



源文件和在源文件中包含的头文件分别为：

```
//----------------------
//     ringx.h
//----------------------
#ifndef RING
#define RING
struct Boy{                        //小孩结构作为链表结点
  int code;
  Boy* next;
}; //--------------------
class Ring{                        //环链表类定义
public:
  Ring(int n, int beg);
  void Count(int m);               //数 m 个小孩
  void PutBoy(bool)const;          //输出当前小孩的编号
  void ClearBoy();                 //将当前小孩从链表中脱钩
  void Display();                  //输出圈中所有小孩
~Ring();
private:
  Boy* pBegin;
  Boy* pivot;
  Boy* pCurrent;
}; //--------------------
#endif

//----------------------
//     josex.h
//----------------------
#ifndef JOSEX
#define JOSEX
class Jose{
public:
  Jose(int boys=10, int begin=1, int m=3){
    numOfBoys=boys;
    beginPos=begin;
    interval=m;
  }
  void Initial();
  void GetWinner();
private:
  int numOfBoys;
  int beginPos;
  int interval;
  int wins;
}; //--------------------
#endif

//----------------------
//     josex.cpp
//----------------------
#include"ringx.h"                  //告诉编译,本文件中将使用Ring
#include"josex.h"
#include<iostream>
```

```
using namespace std;
//----------------------
void Jose::Initial(){
  int num,begin,m,w;
  cout <<"please input the number of boys,\n" \
         "begin position,interval per count :\n" \
         "number of winners :\n";
  cin >> num >> begin >> m >> w;
  if(num < 2){
    cerr <<"bad number of boys\n";
    exit(1);
  }
  if(begin < 0){
    cerr <<"bad begin position.\n";
    exit(1);
  }
  if(m < 1||m > num){
    cerr <<"bad interval number.\n";
    exit(1);
  }
  if(w < 1||w >= num){
    cerr <<"bad number of winners.\n";
    exit(1);
  }
  //输入数据都合法时,予以赋值
  numOfBoys = num;
  beginPos = begin;
  interval = m;
  wins = w;
}//--------------------
void Jose::GetWinner(){
  PutBoy(1);                          //输出从行首开始
  Ring x(numOfBoys, beginPos);        //小孩围成圈
  for(int i = 1; i < numOfBoys - wins + 1; i++){ //处理需要离队的小孩
    x.Count(interval);                //数小孩
    x.PutBoy(0);                      //输出小孩编号
    x.ClearBoy();                     //当前小孩脱离环链
  }
  cout <<"\nthe winner is ";
  x.Count(1);                         //转下一个即胜利者
  x.Display();                        //获胜者
}//--------------------

//----------------------
//    jose5.cpp
//----------------------
#include"josex.h"                     //告诉编译,本文件中将使用Jose类
#include <iostream>
using namespace std;
//----------------------
int main(){
  Jose jose;                          //建立Josephus问题对象
  jose.Initial();
  jose.GetWinner();
}//--------------------
```



运行结果为:

```
please input the number of boys,
begin position,interval per count :
number of winners :
12  1  3  3

    1    2    3    4    5    6    7    8    9 10
   11   12
    3    6    9 12    4    8    1    7    2
the winner is
    5   10   11
```

程序中,用黑体的语句是新添部分。

小结

结构化程序设计强调程序的功能,它以函数(一个功能单位)为中心,分层逐步展开程序设计。但是功能的大小界定,没有统一的标准,不同程序员的设计,反映了不同的思维、程序组织和风格,这给理解程序带来了阴影。同时,它不能隐藏数据复杂性,强迫主程序员必须懂业务、懂计算机以及具有深层次的知识背景。

面向对象程序设计强调程序的分层分类概念,它以抽象为基础,轻灵地描述问题解决的大体思想,以此为基础,进行对象的定义与对象的展示,亦即程序设计,即高层次的抽象程序设计。与此相对应,进行“描述问题解决的大体思想”的具体实现,即类的内部实现(成员函数定义)。

类定义是面向对象程序设计中的基础问题,对象定义是面向对象程序设计的一般操作。定义类的过程要求了解问题中的组织思想和弄清本质。

类是很容易想象的,如果选择了一个错误的类,则描述起来很困难;如果是正确的类,则将很容易理解它,并清楚地描述出其成员函数和数据成员。

面向对象程序设计的关键是如何抽象与分类。在开发大型软件的时候,要用到面向对象的分析设计技术,它不在本书讨论的范围。

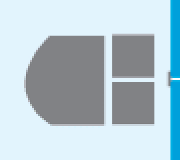
Josephus 问题我们再次提了出来。它在这里用来说明面向对象程序设计和结构化程序设计各自的设计方法,以及其本质的区别。

面向对象程序的维护使我们领略维护并不像结构化程序设计那样大面积展开,而是“各司其职”。

练习

13.1 写出从物质到人类的中间层次。如物质分有机物与无机物,有机物分生物与非生物,生物分动物与植物,等等。

13.2 描述课程类和学生类。用重用类的多文件程序结构形式,编制面向对象应用程序。学生有名字,学生最多可学五门课程,学生实际学的门数,可以给定学生的名字,可以得到学生的名字,可以得到学生给定课程的成绩,可以得到学生所学课程的平均成



绩，可以给学生增加一门课（同时在该课程中增加一个学生）。

课程最多有30个学生，课程有实际学生数，课程有实际学生名单，课程有学分数，课程有每个学生成绩，课程可以得到学分数，课程可以设置学分数，课程可以得到班平均成绩，课程可以得到某个学生成绩。

现有数学课，张三学数学，成绩为3.1分，李四学数学，成绩为4.5分。求其平均成绩，求张三的数学成绩。

现有物理课，学时数为4，张三学物理，成绩为4分。求张三所学课程的平均成绩。

第14章 堆与拷贝构造函数

在 C++中，堆分配的概念得到了扩展，不仅 C++的关键字 new 和 delete 可以分配和释放堆空间，而且通过 new 建立的对象要调用构造函数，通过 delete 删除对象也要调用析构函数。另外，当对象被传递给函数或者对象从函数返回的时候，会发生对象的拷贝，但有些情况，一模一样的拷贝并不是所希望的，这就要借助于定义拷贝构造函数了。学习本章后，应该掌握 new 和 delete 这两个操作符的使用，并能把握从堆中分配和释放对象以及使用对象数组的时机；领会拷贝构造函数的实质，区别浅拷贝和深拷贝，在程序中适当地运用拷贝构造函数。

14.1 关于堆

C++程序的内存格局通常分为 4 个区：

(1) 全局数据区(data area)；

(2) 代码区(code area)；

(3) 堆区(即自由存储区)(heap area)；

(4) 栈区(stack area)。

全局变量、静态数据、常量及字面量存放在全局数据区，所有类成员函数和非成员函数代码存放在代码区，为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区，余下的空间都被作为堆区。

函数“void * malloc(size_t);”和“void free(void *);”在头文件 malloc.h 中声明，而操作符 new 和 delete 是 C++语言的一部分，无须包含头文件。它们都从堆中分配和释放内存块，但在具体操作上两者有很大的区别。

操作堆内存时，如果分配了内存，就有责任回收它，否则运行的程序将会造成内存泄漏。这与函数在栈区分配局部变量有本质的不同。

对 C++来说，管理堆区是一件十分复杂的工作，频繁地分配和释放不同大小的堆空间，将会产生堆内碎块。



14.2 需要 new 和 delete 的原因

从 C++的立场上看，不能用 malloc()函数的一个原因是，它在分配空间的时候不能调用构造函数。类对象的建立是分配空间、构造结构以及初始化的三位一体，它们统一由构造函数来完成。

例如，下面的代码用 malloc()分配对象空间：

```
class Tdate
{
  public:
    Tdate();
    SetDate(int m = 1, int d = 1, int y = 1998);
  protected:
    int month;
    int day;
    int year;
};

Tdate::Tdate()
{
  month = 1;
  day = 1;
  year = 1;
}

void Tdate::SetDate(int m, int d, int y)
{
  if(m > 0 && m < 13)
    month = m;
  if(d > 0 && d < 32)
    day = d;
  if(y > 0 && y < 3000)
    year = y;
}

void fn()
{
  Tdate * pD;                          //仅仅是个指针，没有产生对象
  pD = (Tdate * )malloc(sizeof Tdate); //并不调用构造函数
  //...
  free(pD);                            //并不调用析构函数
}
```

指针 pD 的声明不为 Tdate 调用其构造函数，因为 pD 没有指向任何东西。假如构造函数要被调用，则必须在进行内存分配的 malloc()调用时进行。然而 malloc()仅仅只是一个函数调用，它没有足够的信息来调用一个构造函数，它所接受的参数是一个 unsigned long 类型。



pD 从 malloc()那儿获得的不过是一个含有随机数据的类对象空间而已,对应的对象空间中的值不确定。为此,须在内存分配之后再进行初始化。

例如,下面的代码描述用 malloc()来进行对象的创建过程:

```
void fn()
{
  Tdate * pD;
  pD = (Tdate * )malloc(sizeof Tdate);
  pD -> SetDate();        //设置 Tdate 值
  //...
  free(pD);
}
```

这从根本上说,不是一个类对象的创建,因为它绕过了构造函数。

另外,从程序设计的需要来看,在申请分配内存的时候,必须知道分配的空间派什么用,而且分配空间大小总是某个数据类型(包括类类型)的整数倍。因而 C++用 new 代替 C 的 malloc()是必然的。

14.3 分配堆对象

C++的 new 和 delete 机制更简单易懂。例如,下面的代码可与前面的代码做一比较:

```
void fn()
{
  Tdate * pS;
  pS = new Tdate;       //分配堆空间并构造它
  //...
  delete pS;            //先析构,然后将空间返还给堆
}
```

不必显式指出从 new 返回的指针类型,因为 new 知道要分配对象的类型是 Tdate。而且 new 还必须知道对象的类型,因为它要藉此调用构造函数。

如果是分配局部对象,则在该局部对象退出作用域时(要么程序执行遇到函数结束标记“}”,要么遇到返回语句)自动调用析构函数。但是堆对象的作用域是整个程序生命期,所以除非程序运行完毕,否则堆对象作用域不会到期。堆对象析构是在释放堆对象语句 delete 执行之时。上面的堆对象在执行“delete pS;”语句时,C++自动调用其析构函数。

构造函数可以有参数,所以跟在 new 后面的类类型也可以跟参数。

例如下面的代码,new 后面的类型必须跟参数:

```
class Tdate
{
  public:
    Tdate(int m,int d,int y);
  protected:
    int month;
    int day;
    int year;
};
Tdate::Tdate(int m,ind d,int y)
```

```
{
  if(m>0&&m<13)
    month=m;
  if(d>0&&d<32)
    day=d;
  if(y>0&&y<3000)
    year=y;
}

void fn()
{
  Tdate * pD;
  pD=new Tdate(1,1,1998);
  //...
  delete(pD);
}
```

“pD=new Tdate(1,1,1998);”这一语句，使 new 去调用了构造函数 Tdate(int, int, int)，new 是根据参数匹配的原则来调用构造函数的。如果上一句写成：

```
pD=new Tdate;
```

则由于 Tdate 类没有默认构造函数(已被 Tdate(int,int,int)覆盖)而使该语句报错。

从堆中还可以分配对象数组。

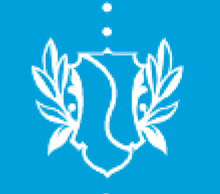
例如，下面的代码分配了参数给定的对象个数，并在函数结束时，予以返还：

```
class Student
{
  public:
    Student(char * pName="no name")
    {
      strcpy(name,pName,sizeof(name));
      name[size of(name)-1]="\0";
    }
  protected:
    char name[40];
};

void fn(int noOfObjects)
{
  Student * pS=new Student[noOfObjects];
  //...
  delete[]pS;
}
```

分配过程将激发 noOfObjects 次构造函数的调用，从 0 至 noOfObjects−1。调用构造函数的顺序依次为 pS[0]，pS[1]，pS[2]，… pS[noOfObjects−1]。由于分配数组时，new 的格式是类型后面跟[元素个数]，不能再跟构造函数参数，所以，**从堆上分配对象数组，只能调用默认的构造函数，不能调用其他任何构造函数**。如果该类没有默认构造函数，则不能分配对象数组。

delete[]pS 中的[]是要告诉 C++，该指针指向的是一个数组。如果在[]中填上了数组



的长度信息,C++编译系统将忽略,并把它作为[]对待。但如果忘了写[],则程序将会产生运行错误。

一般来说,堆空间相对其他内存空间比较空闲,随要随拿,给程序运行带来了较大的自由度。使用堆空间往往由于:

(1) 直到运行时才能知道需要多少对象空间;

(2) 不知道对象的生存期到底有多长;

(3) 直到运行时才知道一个对象需要多少内存空间。

14.4 拷贝构造函数

可用一个对象去构造另一个对象,或者说,用另一个对象值初始化一个新构造的对象,例如:

```
Student s1("Jenny");
Student s2 = s1; //用 s1 的值去初始化 s2
```

对象作为函数参数传递时,也要涉及对象的拷贝,例如:

```
void fn(Student fs)
{
  //...
}

int main()
{
  Student ms;
  fn(ms);
}
```

函数 fn()的参数传递的方式是传值,参数类型是 Student,调用时,实参 ms 传给了形参 fs,ms 在传递的过程中是不会改变的,形参 fs 是 ms 的一个拷贝。这一切是在调用的开始完成的,也就是说,形参 fs 用 ms 的值进行构造。

这时候,调用构造函数 Student(char *)就不合适,新的构造函数的参数应是 Student&,也就是:

```
Student(Student& s);
```

为什么 C++要用上面的拷贝构造函数,而它自己不会做像下面的事呢?即:

```
int a = 5;
int b = a;      //用 a 的值拷贝给新创建的 b
```

因为对象的类型多种多样,不像基本数据类型这么简单,有些对象还申请了系统资源,如图 14-1 所示,s 对象拥有了一个资源,用 s 的值创建一个 t 对象,如果仅仅只是二进制内存空间上的 s 拷贝,那意味着 t 也拥有这个资源了。由于资源归属权不清,将引起资源管理的混乱。在 14.6 节中还要对这个问题展开讨论。

下面的程序介绍了拷贝构造函数的用法:

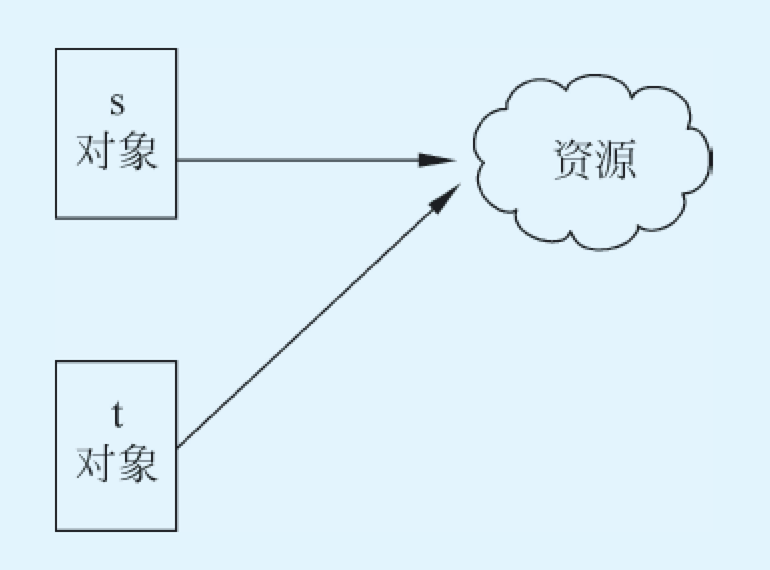


图 14-1　T 对象创建时拷贝 S 对象

```
//----------------------
//    ch14_1.cpp
//----------------------
#include <iostream>
#include <cstring>          //用到 strncpy()、strcat()
using namespace std;
//----------------------
class Student{
  char name[40];
  int id;
public:
  Student(char* pName = "no name", int ssId = 0){
    strncpy(name, pName, 40);
    name[39] = '\0';
    id = ssId;
    cout << "Constructing new student " << pName << endl;
  }
  Student(Student& s){         //拷贝构造函数
    cout << "Constructing copy of " << s.name << endl;
    strcpy(name, "copy of ");
    strcat(name, s.name);
    id = s.id;
  }
~Student(){
    cout << "Destructing " << name << endl;
  }
}; //--------------------
void fn(Student s){
  cout << "In function fn()\n";
}//--------------------
int main(){
  Student randy("Randy", 1234);
  cout << "Calling fn()\n";
  fn(randy);
  cout << "Returned from fn()\n";
}//--------------------
```

运行结果为：

```
Constructing new student Randy
Calling fn()
Constructing copy of Randy
```




```
In function fn()
Destructing copy of Randy
Returned from fn()
Destructing Randy
```

randy 对象的创建调用了普通的构造函数,产生了第一行信息;随之便输出第二行信息;main()调用 fn(randy)时,发生了从实参 randy 到形参 s 的拷贝构造,于是调用拷贝构造函数而得到第三行信息;随之就进入到 fn()的函数体中,产生了第四行信息;从 fn()返回时,形参 s 被析构,所以产生了第五行信息;回到主函数后,输出第六行信息;最后主函数结束时,randy 对象被析构,所以产生了第七行信息。

拷贝构造函数中 strcat()是将 copy of 拼接在 name 之前,它的头文件是 string. h。通常拷贝构造函数将严格限制在只制作拷贝,但是,本程序为了要帮助大家了解其真相,在一些函数体中,设置了输出语句。

头文件 string. h 是 C 语言针对 C 字串操作的头文件,C++对其进行了略微改造而成为 cstring。而头文件 string 是 C++针对 string 字串(本书未涉及)操作的头文件,其又需要 C 字串操作作为基础,所以 string 头文件中又包含了 string. h 头文件。因此,对于 C 字串操作(例如 strncpy)来说,包含 string. h、包含 string、包含 cstring 效果都一样。

C++的 class 定义中,默认访问权限是 private,所以类定义中,若直接说明数据成员或成员函数,则其访问权限就都是 private 的。

14.5 默认拷贝构造函数

类定义中,如果未提供自己的拷贝构造函数,则 C++提供一个默认拷贝构造函数,就像没有提供构造函数时,C++提供默认构造函数一样。

C++提供的默认拷贝构造函数工作的方法是,完成一个成员一个成员的拷贝。如果成员是类对象,则调用其拷贝构造函数或者默认拷贝构造函数。

例如,下面的程序中 Tutor 类使用了默认拷贝构造函数:

```
//---------------------
//    ch14_2.cpp
//---------------------
#include<iostream>
#include<cstring>//用到 strncpy()、strcat()
using namespace std;
//---------------------
class Student{
  char name[40];
public:
  Student(char* pName="no name"){
    cout<<"Constructing new student "<<pName<<endl; strncpy(name,pName,sizeof(name));
    name[sizeof(name)-1]='\0';
  }
  Student(Student& s){
    cout<<"Constructing copy of "<<s.name<<endl; strncpy(name,"copy of ");
    strcat(name,s.name);
  }
```

```
    ~Student(){
      cout <<"Destructing "<< name << endl;
    }
};//--------------------
class Tutor{
  Student student;
public:
  Tutor(Student& s):student(s){
    cout <<"Constructing tutor\n";
  }
};//--------------------
void fn(Tutor tutor){
  cout <<"In function fn()\n";
}//---------------------
int main(){
  Student randy("Randy");
  Tutor tutor(randy);
  cout <<"Calling fn()\n";
  fn(tutor);
  cout <<"Returned from fn()\n";
}//---------------------
```

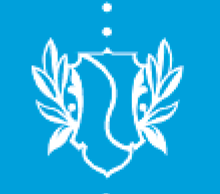
运行结果为：

```
Constructing new student Randy
Constructing copy of Randy
Constructing tutor
Calling fn()
Constructing copy of copy of Randy
In function fn()
Destructing copy of copy of Randy
Returned from fn()
Destructing copy of Randy
Destructing Randy
```

程序一开始运行，进入主函数，首先构造对象 randy，调用 Student 构造函数，产生第一行信息；对象 tutor 是通过调用构造函数 Tutor(Student&)来创建的，该构造函数通过调用 Student 的拷贝构造函数来初始化数据成员 Tutor::student，产生第二行信息；在执行 Tutor 构造函数时，产生第三行信息；接着输出第四行；然后调用 fn()，需要创建 tutor 的一个拷贝，因为 Tutor 类没有定义拷贝构造函数，所以就调用 C++默认的拷贝构造函数，在拷贝成员 student 对象时，调用 Student 拷贝构造函数，结果在名字 copy of Randy 之前又接上了一个 copy of，得到第五行输出；进入 fn()函数体中，得到第六行信息；从 fn()返回时，形参 tutor 析构，调用的是默认析构函数，当析构到成员 student 时，调用 Student 析构函数，产生第七行输出；接着在主函数，输出第八行信息；退出主函数时，先析构 tutor 对象，析构中调用 Student 析构函数，产生第九行信息；最后析构 Randy 对象，得到最后一行输出。

14.6 浅拷贝与深拷贝

在默认拷贝构造函数中，拷贝的策略是逐个成员依次拷贝。但是，一个类可能会拥有资源，当其构造函数分配了一个资源(例如堆内存)的时候，会发生什么呢？如果拷贝构造函数



简单地制作了一个该对象的拷贝,而不对它本身进行资源分配和复制,就得面临一个麻烦的局面:两个对象都拥有同一个资源。当对象析构时,该资源将经历两次资源返还。

例如,下面的程序描述了 Person 对象被简单拷贝后,面临析构时的困惑:

```
//---------------------
//    ch14_3.cpp
//---------------------
#include <iostream>
#include <cstring>  //用到 strcpy()
using namespace std;
//---------------------
class Person{
  char* pName;
public:
  Person(char* pN){
    cout <<"Constructing "<< pN << endl;
    pName = new char[strlen(pN) + 1];
    if(pName!= 0)
      strcpy(pName,pN);
  }
~Person(){
    cout <<"Destructing "<< pName << endl;
    pName[0] = '\0';
    delete pName;
  }
}; //-------------------
int main(){
  Person p1("Randy");
  Person p2 = p1; //即 Person p2(p1);
}//---------------------
```

运行结果为:

```
Constructing Randy
Destructing Randy
Destructing
Null pointer assignment
```

程序开始运行时,创建 p1 对象,p1 对象的构造函数从堆中分配空间并赋给数据成员 pName,同时,产生第一行输出;执行"Person p2=p1;"时,因为没有定义拷贝构造函数,于是就调用默认拷贝构造函数,使得 p2 与 p1 完全一样,并没有新分配堆空间给 p2,见图 14-2;主函数结束时,对象逐个析构,析构 p2 时,将堆中字符串清成空串,然后将堆空间返还给系统,并同时得到第二行输出;析构 p1 时,因为这时 pName 指向的是空串,所以第三行输出中显示的只是 Destructing;当执行"delete pName;"时,系统报错,显示第四行结果。

那还只是某个编译器编译下的代码运行的结果,各编译器生成的堆管理代码并不统一,或许 p1 析构时,在已经释放的堆址上做 pName[0]=0 时就提前崩溃了。

创建 p2 时,对象 p1 被复制给了 p2,但资源并未复制,因此,p1 和 p2 指向同一个资源,这称为浅拷贝。

当一个对象创建时,分配了资源,这时,就需要定义自己的拷贝构造函数,使之不但拷贝成员,也分配和拷贝资源。

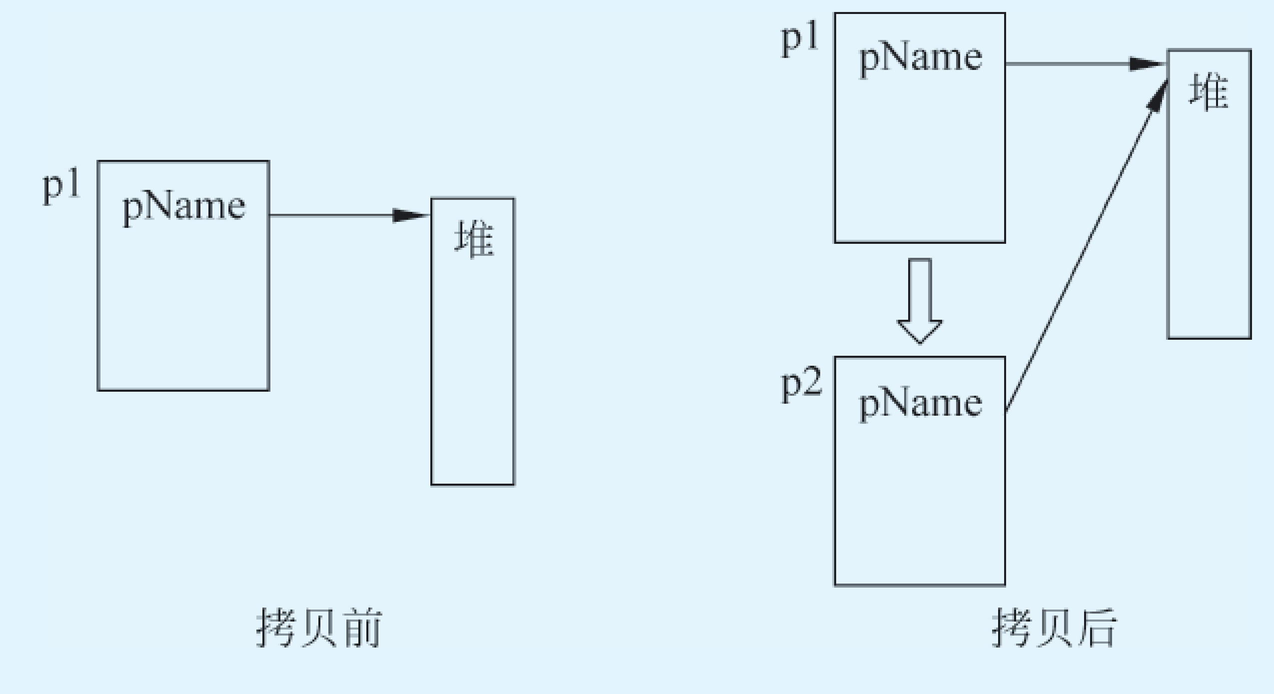


图 14-2 p1～p2 的浅拷贝

例如，下面的代码是在程序 ch14_3.cpp 的基础上，增加一个 Person 类的拷贝构造函数：

```
//----------------------
//  ch14_4.cpp
//----------------------
#include<iostream>
#include<string>
using namespace std;
//----------------------
class Person{
public:
  Person(char* pN);
  Person(Person& p);
  ~Person();
protected:
  char* pName;
}; //--------------------
Person::Person(char* pN){
  cout<<"Constructing "<<pN<<endl;
  pName = new char[strlen(pN) + 1];
  if(pName!= 0)
    strcpy(pName,pN);
}//---------------------
Person::Person(Person& p){
  cout<<"Copying "<<p.pName<<" into its own block\n";
  pName = new char[strlen(p.pName) + 1];
  if(pName!= 0)
    strcpy(pName,p.pName);
}//---------------------
Person::~Person(){
  cout<<"Destructing "<<pName<<endl;
  pName[0] = '\0';
  delete pName;
}//---------------------
int main(){
  Person p1("Randy");
  Person p2(p1);
}//---------------------
```



运行结果为：

```
Constructing Randy
Copying Randy into its own block
Destructing Randy
Destructing Randy
```

程序开始运行时，创建 p1 对象，产生第一行输出；然后用 p1 去创建 p2 对象，调用的是自己定义的拷贝构造函数，于是得到第二行输出；拷贝构造函数中，不但复制了对象空间，也复制资源(堆内存空间)，见图 14-3；当主函数退出时，先后析构 p2 和 p1，但这时候对象们有其各自的资源，所以，析构函数工作得很好，产生最后两行输出。

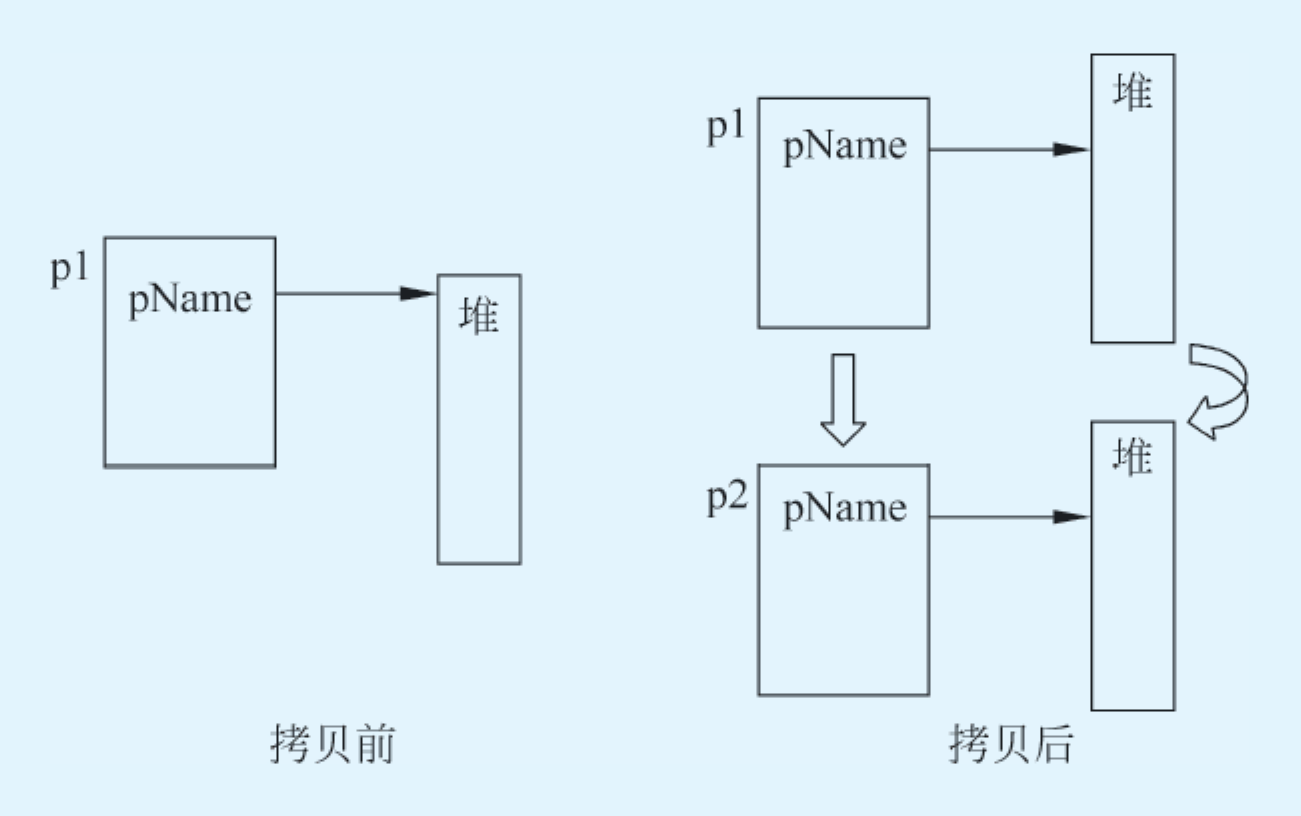


图 14-3　p1～p2 的深拷贝

创建 p2 时，对象 p1 被复制给了 p2，同时资源也作了复制，因此，p1 和 p2 指向不同的资源，这称为深拷贝。

堆内存并不是唯一需要拷贝构造函数的资源，但它是最常用的一个。打开文件，占有硬设备(例如打印机)服务等也需要深拷贝。它们也是析构函数必须返还的资源类型。因此，一个很好的经验是：**如果你的类需要析构函数来析构资源，则它也需要一个拷贝构造函数**。因为通常对象是自动被析构的。如果需要一个自定义的析构函数，那就意味着有额外资源要在对象被析构之前释放。此时，对象的拷贝就不是浅拷贝了。

14.7　临时对象

当函数返回一个对象时，要创建一个临时对象以存放返回的对象。

例如，下面的代码中，返回的 ms 对象将产生一个临时对象：

```
Student fn()
{
  //...
  Student ms("Randy");
  return ms;
}

int main()
{
```

```
    Student s;
    s = fn();
    //...
}
```

在这里,系统调用拷贝构造函数将 ms 拷贝到新创建的临时对象中,见图 14-4。

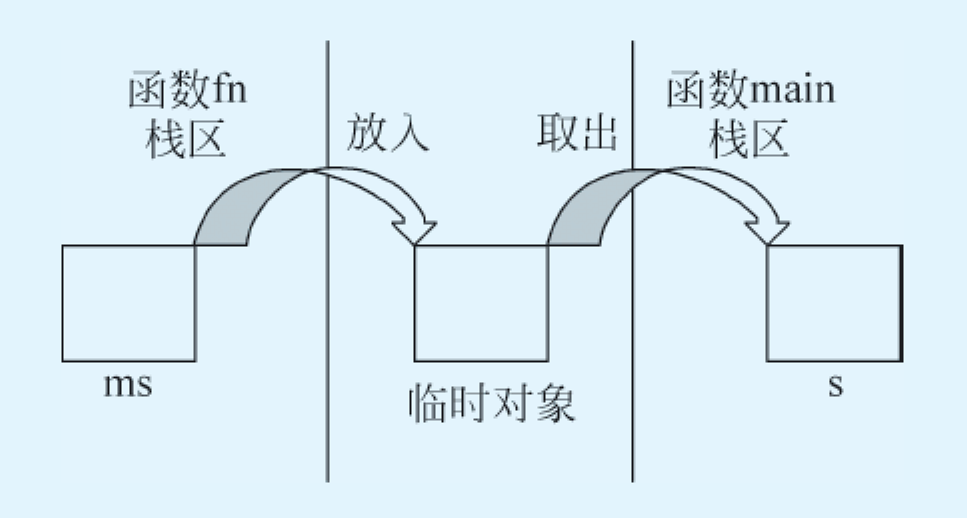


图 14-4　返回对象的函数运行结束时

一般规定,创建的临时对象,在整个创建它们的外部表达式范围内有效,否则无效。也就是说,"s=fn();"这个外部表达式,当 fn()返回时产生的临时对象拷贝给 s 后,临时对象就析构了。

例如,下面的代码中,引用 refs 不再有效:

```
int main()
{
    Student& refs = fn();
    //...
}
```

因为外部表达式"Student& refs=fn();"到分号处结束,以后从 fn()返回的临时对象便不再有效,这就意味着引用 refs 的实体已不存在,所以接下去的任何对 refs 的引用都是错的。

又如,下面的代码中,一切临时对象都在一个外部表达式中结束:

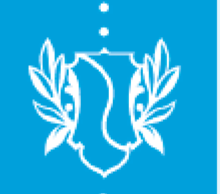
```
Student fn1();
int fn2(Student&);
int main()
{
    int x;
    x = 3 * fn2(fn1()) + 10;
    //...
}
```

fn1()返回时,创建临时对象作为 fn2()的实参,此时,在 fn2()中一直有效;当 fn2()返回一个 int 值参与计算表达式时,那个临时对象仍有效;一旦计算完成,赋值给 x 后,则临时对象被析构。

14.8　无名对象

可以直接调用构造函数产生无名对象。

例如,下面的代码在函数 fn()中,创建了一个无名对象:




```
class Student
{
  public:
    Student(char * );
  //...
};

void fn()
{
  Student("Randy");      //此处为无名对象
 //...
}
```

无名对象可以作为实参传递给函数,还可以拿来拷贝构造一个新对象,也可以初始化一个引用的声明。

例如,下面的代码表达了无名对象典型的三种用法:

```
void fn(Student& s);

int main()
{
  Student& refs = Student("Randy");     //初始化引用
  Student s = Student("Jenny");         //初始化对象定义
  fn(Student("Danny"));                 //函数参数
}
```

主函数开始运行时,第一个执行的是拿无名对象初始化一个引用。由于是在函数内部,所以无名对象作为局部对象产生在栈空间中,从作用域上看,该引用与无名对象是相同的,它完全等价于“Student refs="Randy";”所以这种使用是多余的。

第二个执行的是用无名对象拷贝构造一个对象 s。按理说,C++先调用构造函数“Student(char *);”创建一个无名对象,然后再调用一个拷贝构造函数“Student(Student&);”(或许是默认的)创建对象 s;但是,由于是用无名对象去拷贝构造一个对象,拷贝完后,无名对象就失去了任何作用,对于这种情况,C++特别地将其看作与“Student s="Jenny";”效果一样,而且可以省略创建无名对象这一步。

第三个执行的是无名对象作为实参传递给形参 s,C++先调用构造函数创建一个无名对象,然后将该无名对象初始化给了引用形参 s 对象,由于实参是在主函数中,所以无名对象是在主函数的栈区中创建,函数 fn()的形参 s 引用的是主函数栈空间中的一个对象。它等价于:

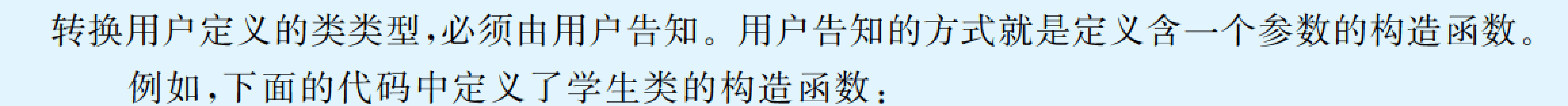
```
Student s("Danny");
fn(s);
```

如果对象 s 仅仅是为了充当函数 fn()实参的需要,完全可以用第三个执行来代替。

当运行到主函数结束的时候,将有一个主函数中的 s 对象和 3 个无名对象被析构。

14.9 构造函数用于类型转换

C++中 5/8 与 5.0/8 结果不同,原因是执行了两种不同的操作。5.0/8 匹配了两个 double 类型数的除法,C++知道如何将 8 转换成 double 型,这是基本数据类型的转换。但是,


转换用户定义的类类型，必须由用户告知。用户告知的方式就是定义含一个参数的构造函数。

例如，下面的代码中定义了学生类的构造函数：

```
class Student
{
  public:
  Student(char * );
  //...
};

void fn(Student& s);

int main()
{
  fn("Jenny");
}
```

这里 Student(char *)构造函数同时也在告知，如何将 char * 转换成一个 Student 对象。如果有重载函数 fn(char *)，则调用 fn("Jenny")马上匹配了事。但就是因为没有这样的重载函数，所以 C++对所有 fn 函数进行类型转换试探，包括构造函数。

因为有 Student(char *)的构造函数，又有 fn(Student& s)函数，于是，fn("Jenny")便被认为是 fn(Student("Jenny"))，最终予以匹配。把构造函数用来从一种类型转换为另一种类型，这是 C++从类机制中获得的附加性能。但要注意下面两点：

(1) 只会尝试含有一个参数的构造函数；

(2) 如果有二义性，则放弃尝试。例如：

```
class Student
{
  public:
    Student(char * pName = "no name");
};

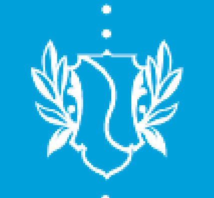
class Teacher
{
  public:
    Teacher(char * pName = "no name");
};

void addCourse(Student& s);
void addCourse(Teacher& t);

int main()
{
  addCourse("Prof.Dingleberry"); //error: 二义性
}
```

改正的方法是，只要显式转换一下：

```
addCourse(Teacher("Prof.Dingleberry"));
```



小结

运算符 new 分配堆内存,如果成功,则返回指向该内存的空间,如果失败,标准 C++则抛出一个 bad_alloc 标准异常,如果捕捉该异常,属于异常编程,见 21.3 节。所以使用运算符 new 动态分配内存时,往往辅之以异常编程。

堆空间的大小是有限的,视其操作系统和编译设置的不同而不同。当程序不再使用所分配的堆空间时,应及时用 delete 释放它们。

由 C++提供的默认拷贝构造函数只是对对象进行浅拷贝复制。如果对象的数据成员包括指向堆空间的指针,就不能使用这种拷贝方式,此时必须自定义拷贝构造函数,为创建的对象分配堆空间。

练习

14.1 阅读下面的程序,写出运行结果。

```
#include <iostream>
using namespace std;
class Samp
{
  public:
    void Setij(int a, int b){i = a,j = b;}

  ~Samp()
    {
      cout <<"Destroying.." << i << endl;
    }

    int GetMulti(){return i * j;}
  protected:
    int i;
    int j;
};

int main()
{
  Samp * p;
  p = new Samp[10];
  if(!p)
  {
    cout <<"Allocation error\n";
    return;
  }

  for(int j = 0; j < 10; j ++ )
    p[j].Setij(j,j);

  for(int k = 0; k < 10; k ++ )
    cout <<"Multi[" << k <<"] is:"
         << p[k].GetMulti() << endl;
  delete[]p;
}
```


14.2 写出下面程序的运行结果，请用增加拷贝构造函数的方法避免存在的问题。

```
 # include < iostream >
 using namespace std;
 class Vector
 {
   public:
     Vector(int s = 100);
     int& Elem(int ndx);
     void Display();
     void Set();
     ~Vector();
   protected:
     int size;
     int * buffer;
};

Vector::Vector(int s)
{
   buffer = new int[size = s];
   for(int i = 0;icsize;i + + )
     buffer[i] = i * i;
}

int& Vector::Elem(int ndx)
{
   if(ndx < 0||ndx >= size)
   {
     cout <<"error in index" << endl;
     exit(1);
   }
   return buffer[ndx];
}

void Vector::Display()
{
   for(int j = 0; j < size; j ++ )
     cout << Elem(j) << endl;
}
void Vector::Set()
{
   for(int j = 0; j < size; j ++ )
     Elem(j) = j + 1;
}

Vector::~Vector()
{
   delete[ ]buffer;
}

int main()
{
   Vector a(10);
   Vector b(a);
   a.Set();
   b.Display();
}
```



14.3 完善下列程序，定义每个成员函数和非成员函数，输出必要的信息，检查临时对象何时被创建，何时被析构。

```
class X
{
  public:
    X(int);
    X(X&);
    ~X();
};

X f(X);

int main()
{
  X a(1);
  X b = f(X(2));
  a = f(a);
}
```

14.4 读下面的程序与运行结果，添上一个拷贝构造函数来完善整个程序。

```
#include <iostream>
using namespace std;
class CAT
{
  public:
    CAT();
    CAT(const& CAT&);
  ~CAT();
    int GetAge() const {return * itsAge;}
    void SetAge(int age){ * itsAge = age;}
  protected:
    int *  itsAge;
};

CAT::CAT()
{
  itsAge = new int;
  * itsAge = 5;
}

CAT::~CAT()
{
  delete itsAge;
  itsAge = 0;
}

int main()
{
  CAT frisky;
  cout <<"frisky's age:" << frisky.GetAge() << endl;
  cout <<"Setting frisky to 6...\n";
  frisky.SetAge(6);
```

```
    cout <<"Creating boots from frisky\n";
    CAT boots(frisky);
    cout <<"frisky's age:" << frisky.GetAge() << endl;
    cout <<"boots'age:" << boots.GetAge() << endl;
    cout <<"setting frisky to 7...\n";
    frisky.SetAge(7);
    cout <<"frisky's age:" << frisky.GetAge() << endl;
    cout <<"boots' age:" << boots.GetAge() << endl;
}
```

运行结果为：

```
frisky's age:5
Setting frisky to 6...
Creating boots from frisky
frisky's age:6
boots' age:6
Setting frisky to 7...
frisky's age:7
boots' age:6
```

第15章 静态成员与友元

类是类型而不是数据对象，每个类的对象都是该类数据成员的拷贝。有时需要让类的所有对象在类的范围内共享某个数据，将所要共享的数据声明为 static 便能在类范围中共享，声明为 static 的类成员称为静态成员。友元函数完全是普通的 C++ 函数，不同的是，它可以访问类的保护成员或私有成员，这样既方便编程，也提高了效率，但破坏了类的封装性。学习本章后，应该掌握怎样声明一个静态数据成员，怎样使用静态成员函数以及静态成员函数为什么与特定对象无关；还应能把握友元的使用，理解友元作用的局限性。

15.1 静态成员的必要性

有一些属性是类中所有对象共有的。

例如 Student 类中，链表的第一个指针以及类中学生数：

```
class Student
{
  //...

  protected:
    Student * pFirst;       //链表首指针
    int count;              //学生数
};
```

这个类声明，意味着每个学生对象都有一个链表首指针和学生数，但是却没有集中成一个成员供所有对象共享。

要想得到现有的学生数，不能到类中去取，因为类不是一个占有内存的实体。那么，到哪个对象中去取？难道一旦学生人数变化，要依次修改每个对象？

如果放在全局变量中，则它们在类的外面，既不安全，又影响了封装性。

例如，下面的代码用全局变量来表示学生类链表首指针和学生人数：

```
class Student
{
```

```
    //...
  };

  int count;                //学生人数
  Student * pFirst;         //学生类链表首指针

  void fn()
  {
    Student ss;             //创建第一个学生对象
    count ++ ;              //学生人数增 1
    pFirst = &ss;           //没有对 pFirst 约束,随便乱用,一点也不把它当链首指针
    //...
  }                         //fn()退出时,ss 作用域终止,ss 被析构,可学生人数忘了减 1
```

面对庞大的程序,没有真正指明哪个函数对 count 和 pFirst 负责。这种无规则会引起软件设计的混乱,一旦程序变大,维护量就急剧上升。

又例如,在程序 ch12_10.cpp 中,定义全局变量 nextStudentId,它既不能放在头文件中定义,也不能放在类 StudentId 的内部实现中,只能放在应用程序主函数 main()的前面。在重用 StudentId 类的时候,总是还要额外地考虑一个全局变量的处置,这不得不使类的封装性受到伤害。

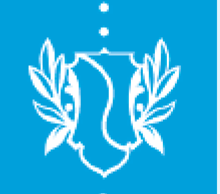
若能将学生人数和链表首指针封装在类里面,不但可以受保护,而且可以作为一个类而重用。这种属于类的一部分,但既不适于用普通成员表示,也不适于用全局变量表示的数据,用静态成员来表示。

15.2 静态成员的使用

类成员有数据成员和成员函数之分,静态成员也有静态数据成员和静态成员函数之分。静态成员用 static 声明。

例如,下面的程序在类中定义了一个静态数据成员和一个静态成员函数,在它的构造函数和析构函数中对静态数据成员进行操作,在应用程序中,调用了静态成员函数:

```
//----------------------
//     ch15_1.cpp
//----------------------
#include <iostream>
#include <string>
using namespace std;
//----------------------
class Student{
  static int noOfStudents;          //C++98 标准版不能写成 noOfStudents = 0;
  char name[40];
public:
  Student(char * pName = "no name"){
    cout <<"create one student\n";
    strncpy(name, pName,40);
    name[39] = '\0';
    noOfStudents++;                 //静态成员:创建对象伴随学生人数增 1
    cout << noOfStudents << endl;
```




```
    }
  ~Student(){
      cout <<"destruct one student\n";
      noOfStudents -- ;                  //析构对象伴随人数减 1
      cout << noOfStudents << endl;
    }
    static int number(){                 //静态成员函数
      return noOfStudents;
    }
}; //--------------------
int Student::noOfStudents = 0;           //静态数据成员初始化
//----------------------
void fn(){
  Student s1;
  Student s2;
  cout << Student::number()<< endl;     //调用静态成员函数用类名引导
}//---------------------
int main(){
  fn();
  cout << Student::number()<< endl;     //调用静态成员函数用类名引导
}//---------------------
```

运行结果为:

```
create one student
1
create one student
2
2
destruct one student
1
destruct one student
0
0
```

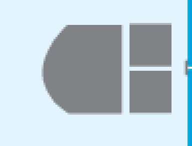
数据成员 noOfStudents,既不是对象 s1 也不是对象 s2 的一部分。Student 类随着对象的产生,每个对象都有一个 name 成员值,但无论对象有多少,甚至没有,静态成员 noOfStudents 也只有一个。所有 Student 对象都能共享它,并且能够访问它。

在 Student 对象空间中,是没有静态数据成员 noOfStudents 的,它的空间,不会随着对象的产生而分配,或随着对象的消失而回收。所以它的空间分配并不在 Student 的构造函数里完成,并且空间回收也不在类的析构函数里完成。

静态数据成员确实是在程序一开始运行时就必须存在。因为函数在程序运行中被调用,所以静态数据成员不能在任何函数内分配空间和初始化。这样,它的空间分配有三个可能的地方,一是作为类的外部接口的头文件,那里有类声明;二是类定义的内部实现,那里有类的成员函数定义;三是应用程序的 main()函数前的全局数据声明和定义处。

静态数据成员要实际地分配空间,故不能在类声明中定义(只能声明数据成员),类声明只声明一个类的“尺寸与规格”,并不进行实际的内存分配,所以在类声明中写成定义“static int noOfStudents=0;”是错误的;它也不能在头文件中类声明的外部定义,因为那会造成在多个使用该类的源文件中,对其重复定义。

静态数据成员也不能在 main()函数之前的全局数据声明处定义,因为那样会使每个重



用该类的应用程序在包含了声明该类的头文件后，都不得不在应用程序中再定义一下该类的静态成员。

例如，下面的代码重用 Student 类，在应用程序中不得不再定义 Student 类的静态成员：

```
file1.cpp                    //Student 类的内部实现部分

#include "student.h"

//类的成员函数定义(没有包括静态数据成员定义)

file2.cpp                    //应用程序重用了 Student 类

#include "student.h"
#include <iostream>
using namespace std;
int Student::noOfStudents = 0;  //不便于重用

void fn()
{
  Student s1;
  Student s2;
  cout << Student::number() << endl;
}

int main()
{
  fn();
  cout << Student::number() << endl;
}
```

静态数据成员是类的一部分，静态数据成员的定义是类定义的一部分，将其放在类的内部实现部分中定义是再合适不过了。定义时要用类名引导。重用该类时，简单地包含其头文件便可。

例如，下面的程序将 ch15_1.cpp 改成了多文件程序实现结构：

```
//---------------------
//    student.h
//---------------------
#ifndef STUDENT
#define STUDENT
class Student{
  static int noOfStudents;
  char name[40];
public:
  Student(char * pName = "no name");
~Student();
  static int number();        //静态成员函数
}; //--------------------
#endif

//---------------------
```




```
//    student.cpp
//----------------------
#include"student.h"
#include<iostream>
#include<string>
using namespace std;
//----------------------
int Student::noOfStudents = 0;      //在此分配空间和初始化
//----------------------
Student::Student(char* pName){
  cout<<"create one student\n";
  strncpy(name, pName, 40);
  name[39] = '\0';
  noOfStudents++;                    //创建对象伴随学生人数增 1
  cout<<noOfStudents<<endl;
}//---------------------
Student::~Student(){
  cout<<"destruct one student\n";
  noOfStudents--;                    //析构对象伴随学生人数减 1
  cout<<noOfStudents<<endl;
}//---------------------
int Student::number(){               //静态成员函数
  return noOfStudents;
}//---------------------

//----------------------
//    ch15_2.cpp
//----------------------
#include"student.h"                  //重用 Student 类
#include<iostream.h>
using namespace std;
//----------------------
void fn(){
  Student s1;
  Student s2;
  cout<<Student::number()<<endl;
}//---------------------
int main(){
  fn();
  cout<<Student::number()<<endl;
}//---------------------
```

工程文件 ch15_2.prj 中包含：

```
//*********************
// ch15_2.prj
//*********************
student.cpp
ch15_2.cpp
//*********************
```

运行结果为：

```
create one student
1
```

```
create one student
2
2
destruct one student
1
destruct one student
0
0
```

15.3 静态数据成员

公共静态数据成员可被类的外部访问，保护或私有静态数据成员只可被类的内部访问。例如，下面的代码描述一个公共的静态数据成员：

```
class Student
{
  public:
    Student()
    {
      noOfStudents ++ ;
      //...
    }
    static int noOfStudents;      //公共静态数据成员
    //...
};

void fn(Student& s1,Student& s2)
{
  cout << s1.noOfStudents;        //此处也可以访问静态数据成员
}
```

在类的外部，访问静态数据成员的形式可以是 s1.noOfStudents，它等价于 s2.noOfStudents，更通常的用法是 Student∷noOfStudents（不能用 Student.noOfStudents）。其意义是，静态数据成员是属于 Student 类的，而不是属于哪个特定对象的，它也不需要依赖某个特定对象的数据。

例如，下面的代码用返回对象引用的成员函数作为对象值去操作静态成员，但是静态成员只取返回对象的类型，其成员函数未被执行：

```
//---------------------
//    ch15_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Student{
public:
  static int noOfStudents;
  Student& nextStudent(){
    noOfStudents++;
    return *this;
  }
  //...
```




```
}; //-------------------
int Student::noOfStudents = 0;
//---------------------
void fn(Student& s){
  cout << s.nextStudent().noOfStudents << endl;  //并未调用 nextStudent()函数
}//--------------------
int main(){
  Student ss;
  fn(ss);
}//--------------------
```

运行结果为:

```
1
```

s.nextStudent()返回 Student 类对象的引用,该引用作为后面的点操作符左操作数,而右操作数是静态数据成员 noOfStudents。

成员函数 nextStudent()不一定会执行(此处被执行),这不是引用成员函数的规范方法,引用静态成员时,C++系统只关心静态成员的类类型。

静态数据成员用得比较多的场合一般为:

(1) 用来保存流动变化的对象个数(如 noOfStudents);

(2) 作为一个标志,指示一个特定的动作是否发生(如可能创建几个对象,每个对象要对某个磁盘文件进行写操作,但显然在同一时间里只允许一个对象写文件,在这种情况下,用户希望说明一个静态数据成员指出文件何时正在使用,何时处于空闲状态);

(3) 一个指向链表第一个成员或最后一个成员的指针(如 pFirst)。

例如,下面的程序描述一个学生类,该类对象是一个个的学生,它们构成一个单向链表:

```
//---------------------
//    ch15_4.cpp
//---------------------
#include <iostream>
#include <string>
using namespace std;
//---------------------
class Student{
  static Student* pFirst;
  Student* pNext;
  char name[40];
public:
  Student(char* pName);
~Student();
}; //-------------------
Student* Student::pFirst = 0;
//---------------------
Student::Student(char* pName){
  strncpy(name,pName,sizeof(name));    //将各种存储区创建的对象都链在一个链表中
  name[sizeof(name) - 1] = '\0';
  pNext = pFirst;                      //每新建一个结点(对象),就将其挂在链首
  pFirst = this;
}//--------------------
Student::~Student(){
```

```
    cout << this -> name << endl;
    if(pFirst == this){          //如果要删除链首结点,则只要链首指针指向下一个
      pFirst = pNext;
      return;
    }
    for(Student * pS = pFirst; pS; pS = pS -> pNext)
      if(pS -> pNext == this){   //找到时,pS 指向当前结点的前结点
        pS -> pNext = pNext;     //pNext 即 this -> pNext
        return;
      }
}//--------------------
Student * fn(){
  Student * pS = new Student("Jenny");
  Student sb("Jone");
  return pS;
}//--------------------
int main(){
  Student sa("Jamsa");
  Student * sb = fn();
  Student sc("Tracey");
  delete sb;
}//--------------------
```

运行结果为:

```
Jone
Jenny
Tracey
Jamsa
```

实现链表结构的学生类,需要一个链首指针 pFirst,每个对象都需要一个指向下一个对象的指针 pNext,所以 pNext 数据成员不是静态的,而 pFirst 数据成员是静态的。

运行时,主函数先创建一个名叫 Jamsa 的学生对象,并使链表含有一个结点,随后调用函数 fn()。在函数 fn()中,从堆中创建一个名叫 Jenny 的学生对象,这时在链中含有 2 个结点,Jenny 是首结点。fn()又在栈中创建 Jone 的学生对象,这时在链中含有 3 个结点,Jone 成为链首结点了。

函数 fn()返回时,名叫 Jone 的 sb 对象的作用域结束,析构它时,将其从链表中删除。删除的是链首结点,此时链中含有 2 个结点。函数 fn()返回一个堆对象的指针给主函数中的 sb。主函数又接着创建一个新的对象 sc,它链入链表中,成为首结点。

最后一条语句释放堆对象空间。释放时,自动析构堆对象,该对象便从链中删除,删除的该结点不在链首,删除过程中,先找到该结点,然后将前后 2 个结点连起来。此后链中剩下 2 个结点。当退出主函数时,先后析构 Tracey 和 Jamsa 的两个对象。析构后,链表清空如初。

链表操作是在构造函数和析构函数中进行的。构造中增加结点的处理比从析构中删除一个结点相对要容易一些。删除一个学生结点时,先要在链表中找到指向当前结点(this 指向的结点)的前结点,然后把当前结点的前结点和后结点链接起来。在实现中,找到当前结点位置时,保存指向当前结点的结点指针是至关重要的。

链表操作的原理见图 15-1、图 15-2。



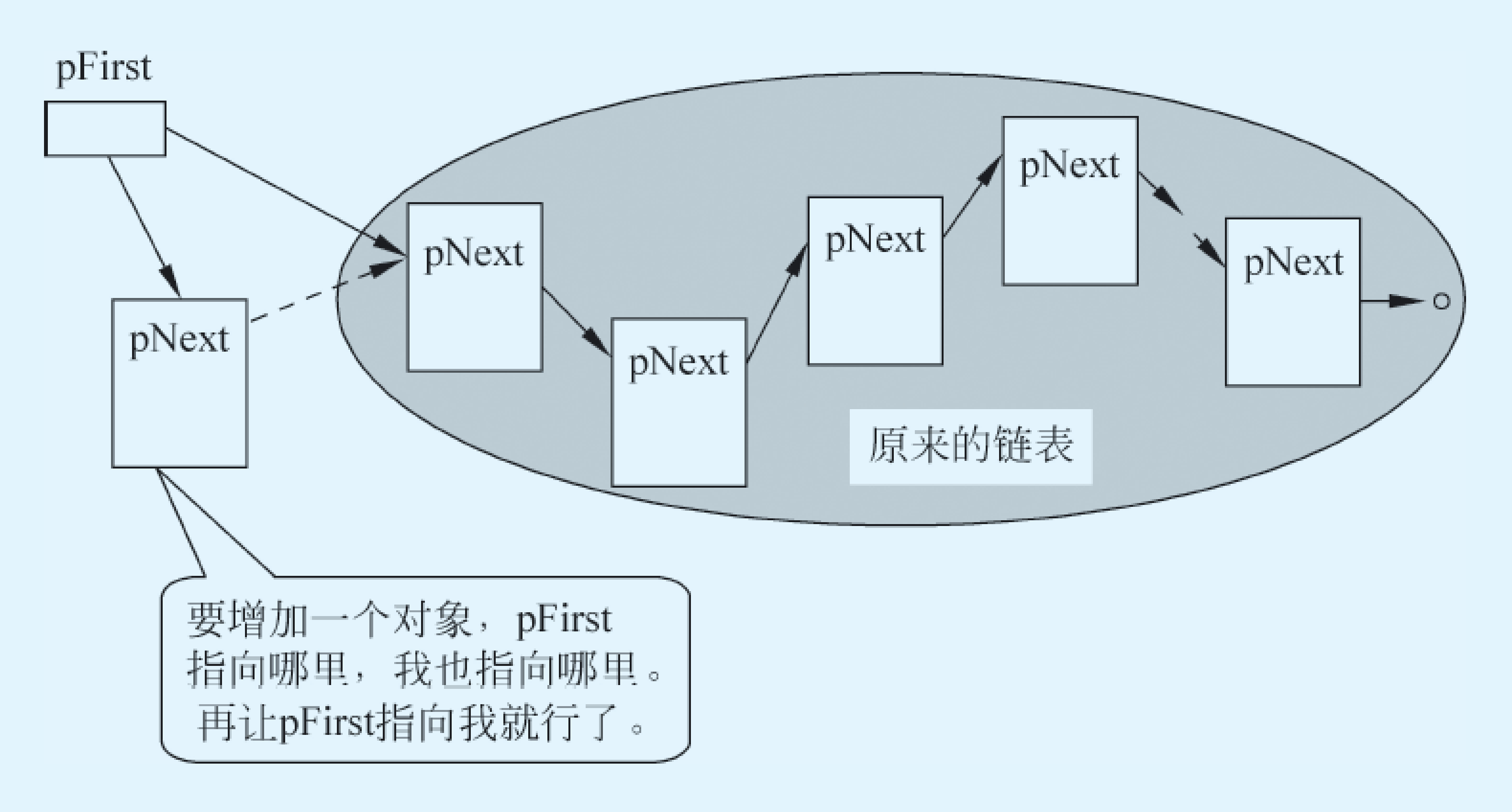


图 15-1　在链表中增加一个学生结点

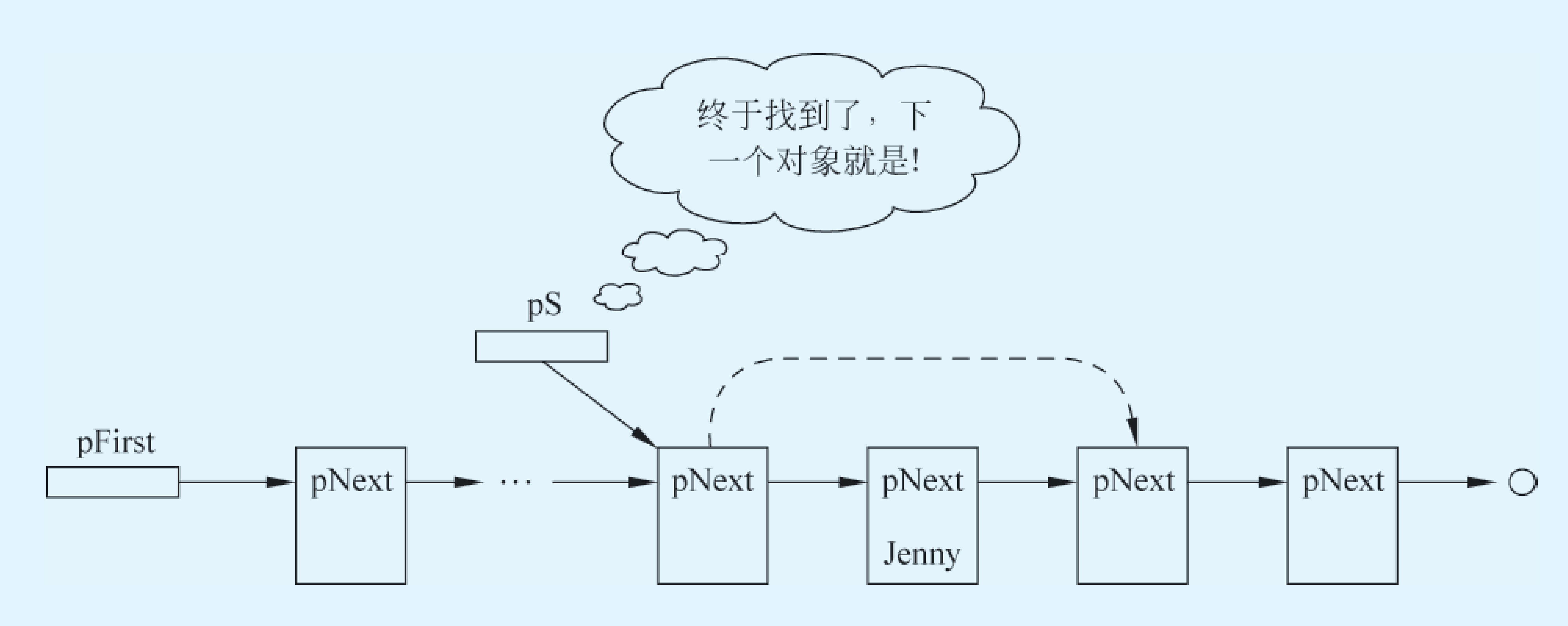


图 15-2　在链表中删除一个学生结点

15.4　静态成员函数

静态成员函数定义在类的内部实现，属于类定义的一部分。它的定义位置与一般成员函数一样。

与静态数据成员一样，静态成员函数与类相联系，不与类的对象相联系，所以访问静态成员函数时，不需要对象引导。如果用对象去引用静态成员函数，只是用其类型。

例如，下面的程序，两种调用静态成员函数的方法都是合法的，而且意义一样：

```
//----------------------
//    ch15_5.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class Student{
  char name[40];
  static int noOfStudents;
public:
  static int number(){
```

```
      return noOfStudents;
    }
    //...
}; //--------------------
int Student::noOfStudents = 1;
//----------------------
int main(){
  Student s;
  cout << s.number()<< endl;           //ok 用对象引用静态成员函数
  cout << Student::number()<< endl;    //ok 用类名引导静态成员函数
}//---------------------
```

运行结果为：

```
1
1
```

一个静态成员函数不与任何对象相联系，故不能对非静态成员进行默认访问。

例如，下面的代码中静态成员函数不应访问非静态数据成员 name：

```
#include <iostream>
using namespace std;
class Student
{
  public:
    static char * sName()      //静态成员函数是所有对象共享的
    {
      cout << noOfStudents << endl;
      return name;              //error 哪个对象?
    }
  protected:
    char name[40];
    static int noOfStudents;
};

int Student::noOfStudents = 0;

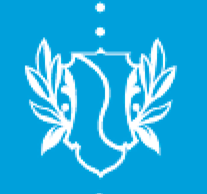
void fn()
{
  //...
  Student s;
  cout << s.sName() << endl;   //sName()从对象 s 上得到的是 Student 类型
}
```

静态成员函数 sName()只认类型，不认对象。即使用对象 s 引导的 s.sName()，也只识别对象 s 所属类型。静态成员的这种性质使得访问任何非静态成员的操作都是非法的。所以在上例的静态成员函数定义中，成员 name 对 sName()来说不知所云。

然而，并不是说静态成员函数不能对非静态成员函数进行访问。

例如，下面的程序说明了一个访问对象中成员的方法：

```
//----------------------
//    ch15_6.cpp
//----------------------
```



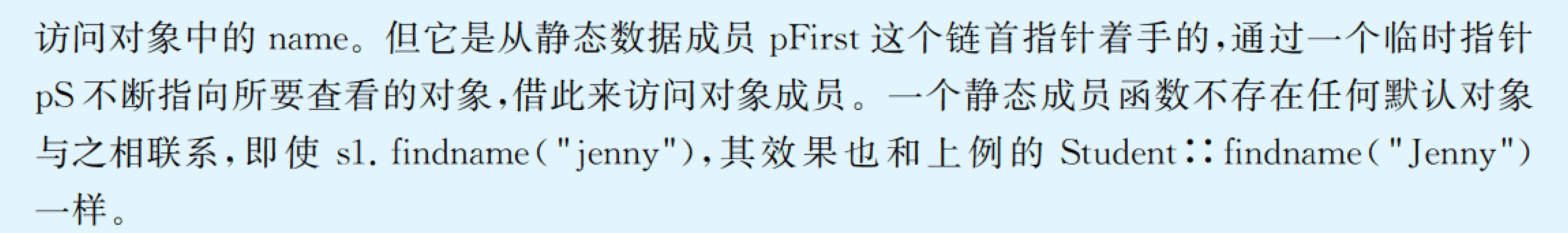

```
#include <iostream>
#include <string>
using namespace std;
//----------------------
class Student{
  static Student* pFirst;
  Student* pNext;
  char name[40];
public:
  Student(char* pName);
  ~Student();
  static Student* findname(char* pName);
}; //--------------------
Student* Student::pFirst = 0; //静态成员空间分配及初始化
//----------------------
Student::Student(char* pName){
  strncpy(name,pName, sizeof(name));
  name[sizeof(name) - 1] = '\0';
  pNext = pFirst;
  pFirst = this;
}//--------------------
Student::~Student(){
  if(pFirst == this){
    pFirst = pNext;
    return;
  }
  for(Student* pS = pFirst; pS; pS = pS->pNext)
    if(pS->pNext == this){
      pS->pNext = pNext;
      return;
    }
}//--------------------
Student* Student::findname(char* pName){
  for(Student* pS = pFirst; pS; pS = pS->pNext)  //通过链表指针来查访对象
    if(strcmp(pS->name,pName) == 0)
      return pS;

  return (Student*)0;
}//--------------------
int main(){
  Student s1("Randy");
  Student s2("Jenny");
  Student s3("Kinsey");
  Student* pS = Student::findname("Jenny");
  if(pS)
    cout<<"ok."<<endl;
  else
    cout<<"no find."<<endl;
}//--------------------
```

运行结果为:

```
ok.
```

findname()是静态成员函数,它查找链中所有对象,看有没有"Jenny"这个学生,因此它要


访问对象中的 name。但它是从静态数据成员 pFirst 这个链首指针着手的，通过一个临时指针 pS 不断指向所要查看的对象，借此来访问对象成员。一个静态成员函数不存在任何默认对象与之相联系，即使 s1. findname("jenny")，其效果也和上例的 Student∷findname("Jenny")一样。

→ 静态成员函数与非静态成员函数的根本区别是什么？

它们的根本区别在于静态成员函数没有 this 指针，而非静态成员函数有一个指向当前对象的指针 this。

例如：

```
class Sc
{
  public:
    void nsfn(int a);           //像声明 Sc::nsfn(Sc * this,int a);
    static void sfn(int a);     //无 this 指针
  //...
}

void f(Sc& s)
{
  s.nsfn(10);                   //转换为 Sc::nsfn(&s, 10)
  s.sfn(10);                    //转换为 Sc::sfn(10)
}
```

函数 nsfn()可被认为它声明为 void Sc∷nsfn(Sc * this,int a)。对 nsfn()的调用，编译像注解的那样进行转换，s 的地址作为第一个传递参数。(你并不实际写该调用，由编译来完成。)

在函数内部，Sc∷nsfn()对非静态成员的访问将自动地把 this 参数作为指向当前对象的指针，而当 Sc∷sfn()被调用时，没有任何对象的地址被传递。因此，当访问非静态成员时，无 this 指针(出错)。这就是为什么一个静态成员函数与任何当前对象都无联系的原因。

15.5 需要友元的原因

有时候，普通函数需要直接访问一个类的保护或私有数据成员。如果没有友元机制，则只能将类的数据成员声明为公共的，从而，任何函数都可以无约束地访问它。

普通函数需要直接访问类的保护或私有数据成员的原因主要是为提高效率。

例如，下面的代码是做矩阵和向量的乘法。矩阵类和向量类分别为 Matrix 和 Vector，乘法操作的函数只能是普通函数，因为一个函数不可能既是这个类的成员又是那个类的成员：

```
//----------------------
//    ch15_7.cpp
//----------------------
#include <iostream>
#include <cstdlib>               //用到 memcpy(), exit()
using namespace std;
//----------------------
class Vector{
  int * v;                       //指向一个数组,表示向量
```




```
    int sz;                  //元素个数
public:
    Vector(int);
    ~Vector(){ delete[]v; }  //将堆中数组空间返还
    Vector(Vector &);
    int Size(){ return sz; }
    void Display();
    int& Elem(int);          //返回向量元素
}; //--------------------
Vector::Vector(int s){
    if(s<=0){
      cerr<<"bad Vector size.\n";
      exit(1);
    }
    sz=s;
    v=new int[s];            //从堆中分配存放向量的数组
}//--------------------
int& Vector::Elem(int i){  //引用返回的目的是返回值可以作左值
    if(i<0||sz<=i){
      cerr<<"Vector index out of range.\n";
      exit(1);
    }
    return v[i];
}//--------------------
Vector::Vector(Vector& vec){
    v=new int[sz=vec.sz];
    memcpy((void*)v,(void*)vec.v,sz*sizeof(int));
}//--------------------
void Vector::Display(){
    for(int i=0; i<sz; i++)
      cout<<v[i]<<" ";
    cout<<endl;
}//--------------------
class Matrix{
    int* m;
    int szl;
    int szr;
public:
    Matrix(int,int);
    Matrix(Matrix &);
    ~Matrix(){ delete[]m; }
    int SizeL(){ return szl; }
    int SizeR(){ return szr; }
    int& Elem(int,int);
}; //--------------------
Matrix::Matrix(int i,int j){
    if(i<=0||j<=0){
      cerr<<"bad Matrix size.\n";
      exit(1);
    }
    szl=i;
    szr=j;
    m=new int[i*j];
}//--------------------
```

```
Matrix::Matrix(Matrix& mat){
  szl = mat.szl;
  szr = mat.szr;
  m = new int[szl * szr];
  memcpy((void * )m,(void * )mat.m, szl * szr * sizeof(int));
}//--------------------
int& Matrix::Elem(int i,int j){          //引用返回
  if(i<0||szl<=i||j<0||szr<=j){
    cerr<<"Matrix index out of range.\n";
    exit(1);
  }
  return m[i * szr + j];
}//--------------------
Vector Multiply(Matrix& m, Vector& v){   //矩阵乘向量的普通函数
  if(m.SizeR()!= v.Size()){
    cerr<<"bad multiply Matrix with Vector.\n";
    exit(1);
  }
  Vector r(m.SizeL());                    //创建一个存放结果的空向量
  for(int i = 0; i<m.SizeL(); i++){
    r.Elem(i) = 0;
    for(int j = 0; j<m.SizeR(); j++)
      r.Elem(i)  += m.Elem(i,j) * v.Elem(j);
  }
  return r;
}//--------------------
int main(){
  Matrix ma(4,3);
  ma.Elem(0,0) = 1; ma.Elem(0,1) = 2; ma.Elem(0,2) = 3;
  ma.Elem(1,0) = 0; ma.Elem(1,1) = 1; ma.Elem(1,2) = 2;
  ma.Elem(2,0) = 1; ma.Elem(2,1) = 1; ma.Elem(2,2) = 3;
  ma.Elem(3,0) = 1; ma.Elem(3,1) = 2; ma.Elem(3,2) = 1;
  Vector ve(3);
  ve.Elem(0) = 2; ve.Elem(1) = 1; ve.Elem(2) = 0;
  Vector va = Multiply(ma,ve);
  va.Display();
}//--------------------
```

运行结果为：

```
 4 1 3 4
```

Matrix 中的 m 和 Vector 中的 v 是保护数据。由于 Multiply()不是 Matrix 和 Vector 类的成员，不能直接操纵 m[i][j]和 v[j]，只能通过 m.Elem(i,j)和 v.Elem(j)来访问矩阵和向量的元素。Elem()函数定义中要对下标进行合法性检查，所以，Multiply()函数要频繁地调用函数和进行下标检查。做一次上例中的小乘法，矩阵(4,3)乘以向量(3)得到向量(4)，其 Elem()和 Size()成员函数要调用 3+1+4 * (1+(1+2) * 3)=44 次，显然效率不高。于是希望乘法不要调用 Elem()函数，能直接访问两个类的保护数据成员。

15.6 友元的使用

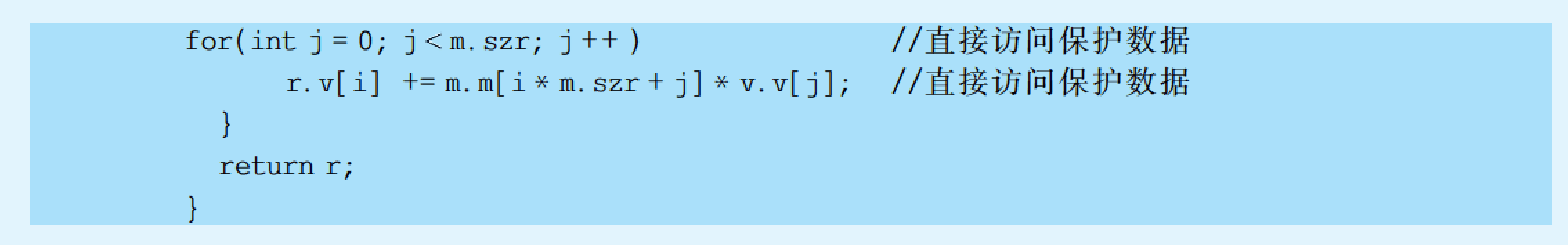
在类里声明一个普通函数,标上关键字 friend,就成了该类的友元,可以访问该类的一切成员。在上节中,若将 Multiply()函数声明为 Matrix 和 Vector 两个类的友元,就能使 Multiply()函数既可访问 Matrix 的保护数据成员,又可访问 Vector 类的保护数据成员了。

例如,下面的代码将程序 ch15_7.cpp 中的 Multiply()改为友元:

```
#include <iostream>
using namespace std;
class Vector
{
  public:
    Vector(int);
    ~Vector(){delete[]v;}
    //int Size(){return sz;} 不需要
    void Display();
    int& Elem(int);
    friend Vector Multiply(Matrix& m, Vector& v);
  protected:
    int* v;
    int sz;
};
class Matrix
{
  public:
    Matrix(int,int);
    ~Matrix(){delete[]m;}
    //int SizeL(){return szl;} 不需要
    //int sizeR(){return szr;} 不需要
    int& Elem(int,int);
    friend Vector Multiply(Matrix& m, Vector& v);
  protected:
    int* m;
    int szl;
    int szr;
};

//此处省略成员函数定义

Vector Multiply(Matrix& m, Vector& v)          //友元定义即普通函数定义
{
  if(m.szr!=v.sz)                               //直接访问保护数据
  {
    cerr<<"bad multiplying Matrix with Vector.\n";
    exit(1);
  }
  Vector r(m.szl);                              //直接访问保护数据
  for(int i=0; i<m.szl; i++)                    //直接访问保护数据
  {
    r.v[i]=0;                                   //直接访问保护数据
```



```
    for(int j = 0; j < m.szr; j++)              //直接访问保护数据
        r.v[i] += m.m[i * m.szr + j] * v.v[j];  //直接访问保护数据
  }
  return r;
}
```

这样一个乘法，由于使用友元直接访问矩阵类和向量类的保护数据，避免了频繁调用成员函数，效率就高多了。

需要友元的另一个原因是为了方便重载操作符的使用。18.2 节和 18.4 节中有关于该内容的介绍。

友元函数不是成员函数，它是类的朋友，因而能够访问类的全部成员。在类的内部，只能声明它的函数原型，加上 friend 关键字。友元声明的位置可在类内部的任何位置，既可在 public 区，也可在 protected 区，意义完全一样。友元函数定义则在类的外部，一般与类的成员函数定义放在一起。因为类重用时，一般友元是一起提供的。

一个类的成员函数可以是另一个类的友元。

例如，下面的代码中，教师应该可以修改学生的成绩（访问学生类的保护数据），将教师类的成员函数 assignGrades()声明为学生类的友元：

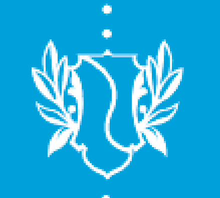
```
class Student;                           //前向声明 类名声明

class Teacher
{
  //...
  public:
    void assignGrades(Student& s);   //给定成绩
  protected:
    int noOfStudents;
    Student * pList[100];
};

class Student
{
  public:
    //...
    friend void Teacher::assignGrades(Student& s);
  protected:
    Teacher * pT;
    int semesterHours;
    float gpa;
};

void Teacher::assignGrades(Student& s)
{
  s.gpa = 4.0;                       //修改学生的平均成绩 gpa
}
```

在教师类声明中，声明了学生类对象的指针数组；在学生类声明中，又声明了教师类对象的指针和作为友元的教师类成员函数。解决这种交叉声明问题的方法是先进行类名声



明。如例中的"class Student;"让编译知道 Student 类的名字已经登记在册,后面可以引用这个名字。类名声明不能用于定义该类的对象,因为这时还没有类的完整声明,没有分配类对象空间的依据(即不能在类名声明"class Student;"后声明 Student 类之前,出现定义类对象语句:Student ss;)。

教师类中的成员函数 assignGrades()需要 Student 类的参数才可以访问参数(Student 类对象)的保护数据。

整个类可以是另一个类的友元,该友元称为友类。友类的每个成员函数都可访问另一个类中的保护或私有数据成员。

例如,下面的代码将整个教师类看成是学生类的友类,教师既可修改成绩,又可调整学时数:

```
class Student;

class Teacher
{
  public:
    void assignGrades(Student& s);    //赋成绩
    void adjustHours(Student& s);     //调整学时数
    //...
  protected:
    int noOfStudent;
    Student * pList[100];
};
class Student
{
  public:
    friend class Teacher;             //友类
    //...
  protected:
    int semesterHours;
    float gpa;
};
```

小结

使用静态数据成员,实际上可以消灭全局变量。全局变量给面向对象程序带来的问题就是违背封装原则。使用静态数据成员必须在 main()程序运行之前分配空间和初始化。使用静态成员函数,可以在实际创建任何对象之前初始化专有的静态数据成员。静态成员不与类的任何特定对象相关联。

静态成员的 static 一词与静态存储类的 static 是两个概念,一个论及类,一个论及内存空间的位置以及作用域限定,所以要区分静态对象和静态成员。

友元的作用主要是为了提高效率和方便编程。在 18.2 节和 18.4 节中还论及了友元在操作符重载中的作用。友元越过了类的封装,直接去操作类的私有成员,带来了性能提升和编程灵活性,加上类的访问权限确实在有些场合比较呆板,还不够完美,所以就容忍了友元这一特别语法现象。



练习

15.1

(1) 编写一个类，声明一个数据成员和一个静态数据成员。让构造函数初始化数据成员，并把静态数据成员加1，让析构函数把静态数据成员减1。

(2) 根据(1)编写一个应用程序，创建三个对象，然后显示它们的数据成员和静态数据成员，再析构每个对象，并显示它们对静态数据成员的影响。

(3) 修改(2)，让静态成员函数访问静态数据成员，并让静态数据成员是保护的。

15.2

(1) 下述代码有何错误，改正它。

```
#include <iostream>
using namespace std;
class Animal;

void SetValue(Animal&, int);
void SetValue(Animal&, int, int);

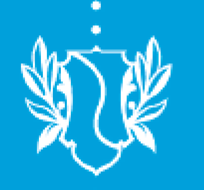
class Animal
{
  public:
    friend void setValue(Animal&, int);
  protected:
    int itsWeight;
    int itsAge;
};

void SetValue(Animal& ta, int tw)
{
  ta.itsWeight = tw;
}
void SetValue(Animal& ta, int tw, int tn)
{
  ta.itsWeught = tw;
  ta.itsAge = tn;
}

int main()
{
  Animal peppy;
  SetValue(peppy, 5);
  SetValue(peppy,7,9);
}
```

(2) 将上面程序中的友元改成普通函数，为此增加访问类中保护数据的成员函数。

15.3 重新编写下述程序，使函数 Leisure()成为类 Car 和类 Boat 的函数。作为重新编程，在类 Car 和类 Boat 中，增加函数 get()。




```
#include <iostream>
using namespace std;
class Boat;

class Car
{
  public:
    Car(int j){size = j;}
    friend int Leisure(int time, Car& aobj, Boat& bobj);
  protected:
    int size;
};
class Boat
{
  public:
    Boat(int j){size = j;}
    friend int Leisure(int time, Car& aobj, Boat& bobj);
  protected:
    int size;
};

int Leisure(int time, Car& aobj, Boat& bobj)
{
  return time * aobj.size * bobj.size;
}
int main()
{
  Car c1(2);
  Boat b1(3);
  int time = 4;
  cout << Leisure(time,c1,b1);
}
```

第16章　继承

继承是 C++语言的一种重要机制，该机制自动地为一个类提供来自另一个类的操作和数据结构，这使得程序员只需定义已有类中没有的成分来建立新类。理解继承是理解面向对象程序设计所有方面的关键。通过学习本章，能利用继承现有的类建立新类，能理解继承如何提高软件的重用性。此外，我们也可以为一个派生类指定多个基类，这样的继承结构被称为多重继承或多继承。应能理解多继承的工作原理，了解多继承要解决的问题，认识虚拟继承的实质，把握多继承的方法，并能简单地从多个基类中派生出新类。

16.1　继承的概念

图 16-1 展示了交通工具的类层次。最顶部的类称为基类，是交通工具类，这个基类有汽车子类，交通工具类是汽车类的父类。可以从交通工具类派生出其他类，比如飞机类、火车类和轮船类，每个类都只有交通工具类作为其父类。汽车子类还有三个子类：小汽车类、旅行车类和卡车类，每个类都以汽车类作为父类，交通工具类可称为它们的祖先类。另外，小汽车类是轿车类、工具车类和面包车类这些派生类的父类。图中展示了小型四层次的类，它用继承来派生子类。每个类有且仅有一个父类，所有子类都是一种父类。例如，小汽车是一种汽车，轿车是一种汽车，汽车是一种交通工具，小汽车也是一种交通工具。这样，每个子类代表父类的特定版本。

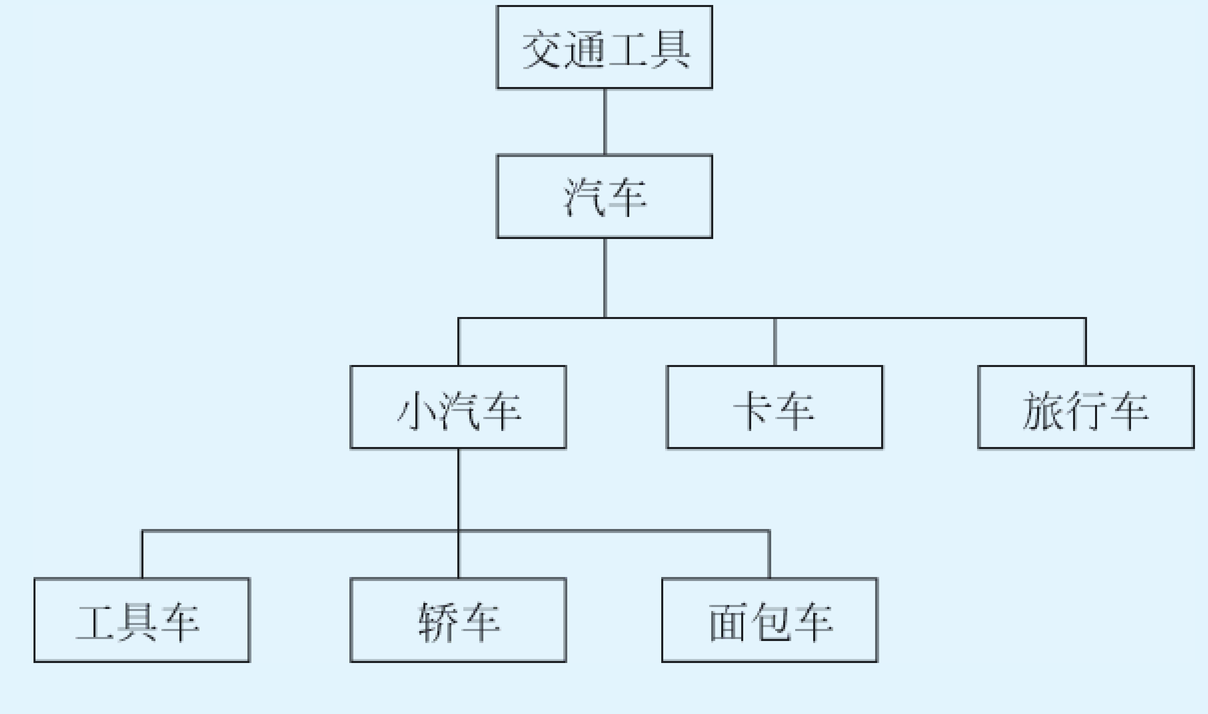
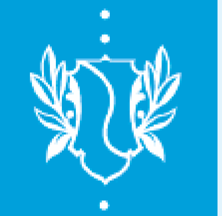


图 16-1　继承的类层次



继承使得我们得以用一种简单的方式来描述事物。比如,描述什么叫鸭子,可以回答说:它是一种嘎嘎叫的鸟。这里鸭子是鸟类的派生,所以鸭子是鸟类的一种,鸭子同时又具有自己的特征,就是会嘎嘎叫,嘎嘎叫是区别于其他鸟类的属性。由于鸟类的特点都清楚,所以用鸟类来描述鸭子,只要举出鸭子自己所具有的特点就行了。继承使我们描述事物的能力大大增强和简单化。

面向对象程序设计可以让你声明一个新类作为另一个类的派生。派生类(也称子类)继承它父类的属性和操作。子类也声明了新的属性和新的操作,剔除了那些不适合于其用途的继承下来的操作。这样,继承可让你重用父类的代码,专注于为子类编写新代码。

当父类已经存在,在新的应用中你无须修改父类,所要做的就是派生子类,在子类中做一些增加与修改。这种机制,使重用成为可能。

早些时候在标准 C 函数库中,很少能找到可重用的软件部件。如果一个程序员已经开发了一些程序,现在要开发一个新的程序,实际上不可能在先前的程序中找到完全符合要求的程序部件,通常这些部件需要调整修改。

现实世界是分类分层的客观实在,物质有无机与有机之分,有机体有生命体与非生命体之分,生命体有动物、植物、微生物等,动物有各种,人是其中的一种,人有各个种族……

继承是我们理解事物、解决问题的方法。继承帮助我们描述事物的层次关系,帮助我们精确地描述事物,帮助我们理解事物的本质。一旦看清了事物所处的层次结构,也就找到了对应的解决办法。

继承可以使已存在的类无须修改地适应新应用,理解继承是理解面向对象程序设计所有方面的关键。

16.2 继承的工作方式

现有一个大学生类 Student,要增加研究生类 GraduateStudent。由于研究生除了他自己特有的性质外,具有大学生的所有性质,所以我们用继承的方法来重用大学生类。

```
class Student
{
  //...
};

class GraduateStudent :public Student
{
  //...
};
```

在这里,一个 GraduateStudent 类继承了 Student 类的所有成员。继承的方式是在类定义中类名后跟:public Student。一个研究生是一个大学生,当然,研究生类 GraduateStudent 也包含有自己特有的成员。

下例是继承 Student 的例子,给它添加一些成员:

```
//----------------------
//    ch16_1.cpp
//----------------------
#include <iostream>
#include <string>                        //用到strncpy()
using namespace std;
//----------------------
class Advisor{
  int noOfMeeting;
}; //--------------------
class Student{
  char name[40];
  int semesterHours;
  float average;
public:
  Student(char* pName = "no name"){
    strncpy(name, pName, sizeof(name));
    average = semesterHours = 0;
  }
  void addCourse(int hours, float grade){
    average = (semesterHours * average + grade); //总分
    semesterHours += hours;                      //总修学时
    average /= semesterHours;                    //平均分
  }
  int getHours(){ return semesterHours; }
  float getAverage(){ return average; }
  void display(){
    cout <<"name = \""<< name <<"\", hours = "<< semesterHours <<", average = "<< average << endl;
  }
}; //--------------------
class GraduateStudent : public Student{
  Advisor advisor;
  int qualifierGrade;
public:
  getQualifier(){ return qualifierGrade; }
}; //--------------------
int main(){
  Student ds("Lo lee undergrade");
  GraduateStudent gs;
  ds.addCourse (3, 2.5);
  ds.display();
  gs.addCourse(3, 3.0);
  gs.display();
}//--------------------
```

运行结果为：

```
name = "Lo lee undergrade", hours = 3, average = 0.833333
name = "no name", hours = 3, average = 1
```

ds是Student类对象，gs是GraduateStudent类对象。作为Student的子类，对象gs可以做ds能做的任何事情，它有name，semesterHours，average数据成员，以及addCourse()成员函数，此外它还比ds多一点东西，即它有导师Advisor和资格考试分qualifierGrade。

gs当然也是一个大学生，所以对Student中的addCourse()成员函数的调用，等于是在

调用自己的成员函数。正是由于 gs 是一个大学生,所以,对下面的代码:

```
void fn(Student& s)
{
  //任何学生想要做的事
}

int main()
{
  GraduateStudent gs;
  fn(gs);
}
```

函数 fn()期望接受 Student 类对象作为它的参数,来自 main()的调用传给它一个 GraduateStudent 对象,fn()把它视同 Student 对象予以接受。

事实上,GraduateStudent 的内存布局,也与"gs 是研究生,当然也是大学生"相吻合。见图 16-2。

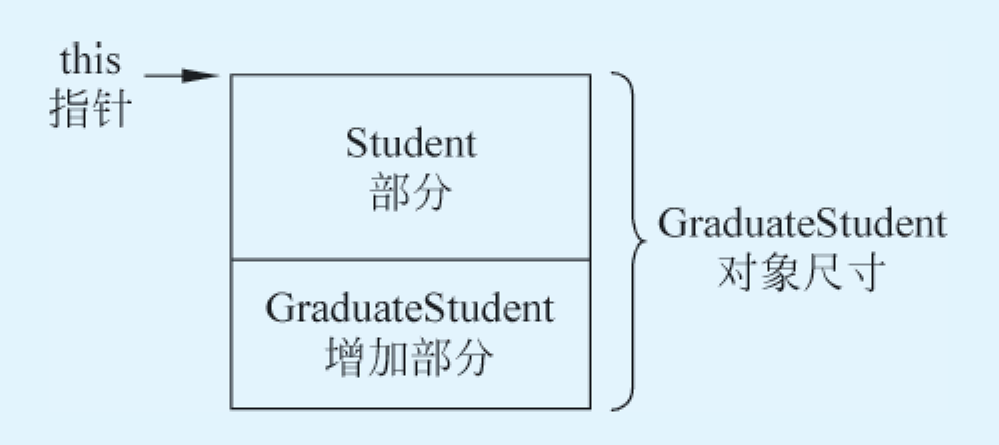


图 16-2　GraduateStudent 内存布局

在布局上 gs 对象的最初部分是 Student 数据成员,fn(gs)等于做了一个 Student(gs)的隐式转换,指向 gs 的 this 指针也就是指向 Student 对象的指针。由此看出,gs 对象中包含有 Student 对象空间部分,gs 用 this 指针访问 Student 成员与访问自己增加的成员没有差别。

16.3　派生类的构造

在 ch16_1.cpp 的例子中,并没有声明派生类 GraduateStudent 的构造函数,根据类的实现机制,派生类对象创建时,将执行其默认的构造函数。该默认构造函数会首先调用基类的默认构造函数,而基类没有默认构造函数,但正好匹配默认参数的构造函数。所以在运行结果中,gs 对象的 name 值为"no name"。

派生类自己的成员函数可以直接访问基类的保护数据成员,甚至在构造时初始化它们,但是一般不这么做,而是通过基类的接口(成员函数)去访问它们,初始化也是通过基类的构造函数。这样做的好处是,一旦基类的实现有错误,只要不涉及接口(基类成员函数的声明),那么基类的修改不会影响派生类的操作。**类与类之间,你做你的,我做我的,以接口作沟通。即使基类与子类也不例外**。这正是类能够发挥其生命力的原因所在。

在构造一个子类时,完成其基类部分的构造由基类的构造函数去做,C++类的继承机制满意地提供了这种支持。

例如,下面的代码定义了研究生类,其中的构造函数实现对基类数据成员的构造:

```
class GraduateStudent :public Student
{
  public:
    GraduateStudent(char * pName, Advisor& adv)
                :Student(pName), advisor(adv)
    {
      qualifierGrade = 0;
    }
    //其余见 ch16_1.cpp
};

void fn(Advisor& advisor)
{
  GraduateStudent gs("Yen Kay Doodle", advisor);
  //...
}

int main()
{
  Advisor da;
  fn(da);
}
```

main()函数中创建了 Advisor 对象，以此为参数调用了函数 fn()。fn()中创建了 GraduateStudent 对象，初始化的参数为“Yen Kay Doodle”和 advisor。在构造函数原型的后面是 Student(pName)，advisor(adv)，表示对数据成员初始化的方式。

基类初始化由 Student(pName)去完成，派生类的构造总是由基类的初始化开始的。由图 16-2 我们看到，基类在派生类对象中的空间位置也是如此。如果在上面初始化的方式中，描述顺序为：advisor(adv)，Student(pName)，那么，派生类构造开始时，仍然是先调用基类的构造函数。如果 Student(pName)没有，则派生类会调用基类的默认构造函数，如果找不到匹配的构造函数，则就通不过编译。此处调用的构造函数是以 pName 为参数，而 pName 就是创建派生类对象时，由应用程序传递而来，它是派生类构造函数的形参。

上述代码中，由于 GraduateStudent 类参数 pName 为“Yen Kay Doodle”，所以基类的构造函数调用 Student(pName)产生一个“Yen Kay Doodle”的 Student 对象。派生类构造函数启动时，首先量好了派生类对象的大小尺寸，规定了对象分配的位置(给 this 赋值)，于是后面基类构造函数所产生的对象就构造在这个 this 指针指向处。

advisor 是 GraduateStudent 的数据成员，它是 Advisor 类对象。advisor(adv)表示以 adv 的值来初始化 advisor。因为参数 adv 为 Advisor 对象，所以 advisor(adv)将调用 Advisor 的默认拷贝构造函数(Advisor 自己没有定义拷贝构造函数之故)。

在派生类构造函数体内，“qualifierGrade =0;”完成对其数据成员的赋值。

派生类的析构函数以构造函数相反的顺序被调用，所以函数 fn()返回时，系统开始处理 gs 对象的析构，先调用 GraduateStudent 的析构函数，执行析构函数体的代码，然后调用 Advisor 析构函数，最后调用 Student 析构函数。

创建 fn()中 gs 对象时，参数 advisor 与 gs 对象中的数据成员 advisor 处在两个不同的对象空间，所以当 fn()函数返回时，析构 gs 之后，并没有把函数 fn()的形参 advisor 析构掉，由于该形参是传引用方式，所以 fn()返回时只是解除引用关系，对其不做其他处理。advisor 形参就是

main()函数中的 Advisor 对象 da。一直到 main()函数结束时,才由 Advisor 析构函数将 da 析构。

16.4 继承方式

1. 保护成员与私有成员的区别

类有公有、保护和私有 3 种访问权限。当一个类作为基类时,对于使用基类的应用编程,其在类作用域之外,只能访问基类的公有成员,不能访问基类的保护成员和私有成员;对于公有继承基类的类编程,其在派生类作用域中,只能访问基类的公有成员和保护成员,不能访问基类的私有成员。这便是保护成员与私有成员的区别。也即保护成员和私有成员对于应用编程来说,无任何差别;对于类编程,则保护成员对派生类网开一面,保护成员专为继承而设。例如:

```
class Base{
  int b1;
protected:
  void fb2(){ b1 = 1; }
public:
  void fb3(){ b1 = 2; }
};

class Pub : public Base{        //公有继承
public:
  void test(){
    b1 = 1;                     //error
    fb2();                      //ok
    fb3();                      //ok
  }
};

int main(){
  Base b;
  b.fb2();                      //error
  b.fb3();                      //ok
}
```

2. 保护继承与私有继承

继承可以公共继承,也可保护继承和私有继承。其形式为:

```
class Base{
  //…
};

class Pri : private Base{          //私有继承
  //…
};

class Pro : protected Base{        //保护继承
  //…
};
```

保护和私有继承这种代码重用方式更多的是考虑到代码安全，初学者很少涉及。

如果是公共继承，将使基类作为开源代码不断让类程序员派生代码从而迅速传播。

如果是保护继承，那么将使基类作为公司内部技术不断接续和派生。

如果是私有继承，那么将使技术世代单传，开发者在其派生类中也只能通过调用基类的成员函数来访问基类。

一个私有的或保护的派生类不是子类，因为非公共的派生类不能做基类能做的所有事。例如，下面的代码定义了一个私有继承的基类：

```
#include<iostream>
using namespace std;

class Animal
{
public:
  Animal(){}
  void eat(){ cout<<"eat\n"; }
};

class Giraffe : private Animal
{
public:
  Giraffe(){}
  void StrechNeck(double){ cout<<"strech neck\n"; }
};

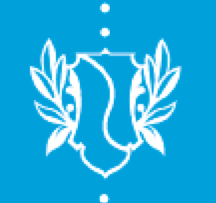
class Cat : public Animal
{
public:
  Cat(){}
  void Meaw(){ cout<<"meaw\n"; }      //喵喵叫
};

void Func(Animal& an)
{
  an.eat();
}

int main()
{
  Cat dao;
  Giraffe gir;
  Func(dao);
  Func(gir);                        //error
}
```

函数 Func()要用一个 Animal 类型的对象，但调用 Func(dao)实际上传递的是 Cat 类的对象。因为 Cat 公共继承了 Animal 类，所以 Cat 类对象拥有 Animal 的所有成员的使用。Animal 对象可以做的事，Cat 对象也可以做。

但是，对于 gir 对象就不一样了。Giraffe 类私有继承了 Animal 类，意味着对象 gir 不能直接访问 Animal 类的成员。其实，在 gir 对象空间中，包含 Animal 类的对象，只是无法



让其公开访问。

公有继承就像是三口之家的小孩,饱受父母的关爱,享有父母的一切(public 和 protected 的成员)。其中保护的成员不能被外界所享有,但可以为小孩所拥有。只是父母还是有一点点隐私(private 成员)不能为小孩所知道。

私有继承就像是离家出走的小孩,一个人在外面飘泊。他(她)不能拥有父母的住房和财产(如 gir.eat()是非法的),在外面自然也就不能代表其父母,甚至他(她)不算是其父母的小孩(如 Func(gir)调用被禁)。但是在他(她)的身体中,毕竟流淌着父母的血液(在 gir 对象空间中,含有 Animal 对象),所以,在小孩自己的行为中又有着与父母相似的成分(可以通过自身成员函数访问父类私有成员)。

例如,下面的代码中,Giraffe 继承了 Animal 类,Giraffe 的成员函数可以像 Animal 对象那样访问其 Animal 成员:

```
//---------------------
//   ch16_2.cpp
//---------------------
#include<iostream>
using namespace std;
//---------------------
class Animal{
public:
  Animal(){}
  void eat(){ cout<<"eat.\n"; }
}; //-------------------
class Giraffe :private Animal{
public:
  Giraffe(){}
  void StretchNeck(){ cout<<"stretch neck.\n"; }
  void take(){ eat(); }            //ok
}; //-------------------
void Func(Giraffe & an){
  an.take();
}//---------------------
int main(){
  Giraffe gir;
  gir.StretchNeck();
  Func(gir);                       //ok
}//---------------------
```

运行结果为:

```
stretch neck.
eat.
```

上例中,gir 对象就好比是小孩;eat()成员函数是其父母的行为;take()成员函数是小孩的行为,在该行为中,渗透着父母的行为。但是小孩无法直接使用 eat()成员函数,因为离家出走的他(她)无法拥有其父母的权力。

保护继承与私有继承类似,继承之后的类相对于基类来说是独立的,保护继承的类对象,在公开场合同样不能使用基类的成员:

```
#include<iostream>
using namespace std;

class Animal
{
  public:
    Animal(){}
    void eat(){ cout<<"eat\n"; }
};

class Giraffe : private Animal
{
  public:
    Giraffe(){}
    void StrechNeck(double){ cout<<strech neck\n"; }
    void take()
    {
       eat();          //ok
    }
};

int main()
{
  Giraffe gir;
  gir.eat();          //error
  gir.take();         //ok
  gir.StretchNeck();
}
```

3. 访问控制

对于不同的继承方式,其访问控制的约束是不同的。表 16-1 列出了其中的区别。

表 16-1　基类成员在派生类中的访问控制属性

基类访问属性		public	protected	private
继承类型	public	public	protected	隔离
	protected	protected	protected	隔离
	private	private	private	隔离

公有继承将基类的保护成员和公有成员视为自己的保护和公有成员。

保护继承将基类的保护成员和公有成员全变为自己的保护成员。

私有继承将基类的保护成员和公有成员全变为自己的私有成员。

基类的私有成员在派生类采用任何继承方式下都是隔离的,也就是视派生类为“外人”,必须通过基类的保护成员或公有成员函数去访问它们。

基类的公有成员在派生类中的访问属性随继承方式而定。即公有继承下为公有成员,保护继承下为保护成员,私有继承下为私有成员。

例如,对于私有继承,其派生类的成员函数可以访问基类保护和公有成员,却不能直接访问基类的私有成员,必须通过保护成员或公有的基类成员函数去访问基类的私有成员。

```
class Base
{
  int b1;
protected:
  void fb2(){ b1 = 1; }
public:
  int b3;
};

class Pri : private Base
{
public:
  void test(){
    b1 = 1;       //error:基类私有数据被隔离
    b3 = 3;       //ok:将公有成员变为自己的私有成员
    fb2();        //ok:通过可以访问的基类成员函数去访问隔离成员
  }
};
```

4. 调整访问控制

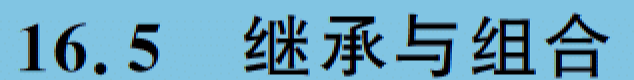
在派生类中,可以调整成员的访问控制属性。例如,可以将公有成员调整为私有成员,将保护成员调整为公有成员,等等。

```
class Base{
  int b1;
protected:
  int b2;
  void fb2(){ b1 = 1; }
public:
  int b3;
  void fb3(){ b1 = 1; }
};

class Pri : private Base{    //私有继承致全部成员为私有或不可见
public:
  using Base::b3;            //抽调其中的可见成员 b3 为公有
};

int main()
{
  Pri pri;
  pri.b3 = 1;                // ok
}
```

调整访问控制属性的前提是在派生类中该成员必须是可见的。例如,上述代码中的私有成员 b1,不管 public、protected、private 的哪种继承方式,它都是不可见的。在派生类中要访问它必须通过基类的保护或公有成员函数,因此 b1 就无法在派生类中进行访问属性的调整,它在子孙类中永远是不可见的。



16.5 继承与组合

1. 继承与组合的概念

类以另一个类对象作数据成员，称为组合。如程序 ch16_1.cpp 中，GraduateStudent 类组合了 Advisor 类。这种场合，称 GraduateStudent 有一个 Advisor；而在继承的场合，称 GraduateStudent 是一个 Student。继承和组合都发挥了作用，它们将以前设计好的类采用"拿来主义"，但二者在使用上不同。GraduateStudent 类是 Student 类的一种，所以 GraduateStudent 享有 Student 的一切待遇。Advisor 含在 GraduateStudent 中，不是继承关系，Advisor 并不从属于 GraduateStudent。这种关系上的差别决定了操作上的差别。例如下面的代码中，定义了一个派生类 Car，该类中包含类对象成员 motor：

```
class Vehicle
{
  //...
};
class Motor
{
  //...
};

class Car : public Vehicle
{
public:
  Motor motor;
};

void vehicleFn(Vehicle& v);
void motorFn(Motor& m);

int main()
{
  Car c;
  vehicleFn(c);           //ok
  motorFn(c);             //error
  motorFn(c.motor);       //ok
}
```

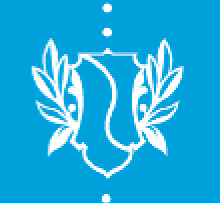
汽车是车辆的子类，它继承了车辆的所有特征。而汽车具有马达，如果拥有了一辆汽车，因为汽车包含马达，所以汽车也拥有了一个马达。

调用 vehicleFn(c)是允许的，因为参数要求是车辆，而汽车是车辆的一种。

调用 motorFn(c)是不允许的，因为参数要求是马达，而汽车不是马达，所以参数 c 不能匹配该函数。

调用 motorFn(c.motor)是允许的，因为参数要求是马达，而汽车中包含的马达就是 c.motor。

采用继承方式还是组合方式，要看类层次的描述，需要分析对象之间的关系。例如，汽车与马达的关系，更多的是包含关系，所以应该设计成汽车组合马达更合适。而小轿车作为



车辆的一种,更多的是分类关系,所以应该设计成车辆派生小轿车更合适。而在技术上,无论组合设计还是继承设计,都能实现所需功能。

2. 继承设计

对于几何图形来说,圆类与直角坐标的点类,关系似乎不是很明确,若从一切图形出自坐标点的意义上看,圆类可以继承自点类。于是对于一个坐标点类:

```
//point.h 直角坐标点类头文件

#include<iostream>
using namespace std;
class Point{
protected:
  double x, y;
public:
  Point(double a = 0, double b = 0):x(a),y(b){}
  double xOffset()const{ return x; }
  double yOffset()const{ return y; }
  void moveTo(double a, double b){ x = a, y = b; }
  void print(){ cout<<"("<<x<<","<<y<<")\n"; }
};
```

该坐标点类的两个分量设计成保护属性,是希望作为基类继承其他几何图形。除了构造函数,还包含获取 x、y 分量的 xOffset()和 yOffset(),修改坐标点位置的 moveTo()以及输出坐标点的 print()成员函数。

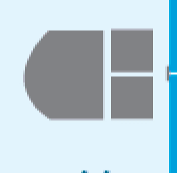
在此基础上,可以继承一个圆类:

```
//circle.h  圆类继承点类头文件

#include"point.h"

class Circle : public Point{
  double radius;
public:
  Circle(const Point& p, double r):Point(p),radius(r){}
  double getRadius()const{ return radius; }
  Point getPoint()const{ return *(Point*)this; }
  double getArea()const{ return radius * radius * 3.14; }
  double getCircum()const{ return 2 * radius * 3.14; }
  void moveTo(double a, double b){ x = a, y = b; }
  void modifyRadius(double r){ radius = r; }
};
```

其中成员函数 getPoint 是返回圆心的坐标,即返回基类的点坐标。因为基类对象位于子类对象之首,子类地址即基类地址,所以取自子类对象地址,作一类型转换就能获得基类点对象地址,然后间接访问之即为坐标。而对于圆的 moveTo 成员函数,修改圆心的 x、y 坐标,可以直接修改基类的 x、y 这两个保护成员,当然也可以调用基类成员函数 Point::moveTo(a,b)来完成。



3. 组合设计

还是对应几何图形来说,圆类也可以把圆心坐标点看成是其组成部分。圆类包含圆心和半径这两个数据成员。这时候可以按组合设计来描绘圆类。

```
//circle.h 圆类组合点类头文件

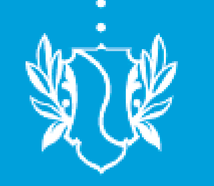
#include"point.h"

class Circle{
   Point point;
   double radius;
public:
   Circle(const Point& p, double r):point(p),radius(r){}
   double getRadius()const{ return radius; }
   Point getPoint()const{ return point; }
   double getArea()const{ return radius * radius * 3.14; }
   double getCircum()const{ return 2 * radius * 3.14; }
   void moveTo(double a, double b){ point.moveTo(a, b); }
   void modifyRadius(double r){ radius = r; }
};
```

注意圆类构造函数的变化,成员构造和基类构造的不同在于,前者是以成员的名义去初始化,后者是调用基类构造函数,完成基类对象构造。至于成员函数 getPoint 直接返回成员值就行了。而对于成员函数 moveTo,也是涉及成员的操作,调用其成员函数即可。

圆类设计中,无论继承还是组合,其公有的成员函数都是一样的,所以对应用程序设计就没有影响了,只要包含对应的圆类头文件,可以得到同样的运行结果。下列代码 ch16_3.cpp,采用圆类的组合策略。

```
//----------------------
// ch16_3  圆类组合策略
//----------------------
#include<iostream>
using namespace std;
//----------------------
class Point{
protected:
   double x, y;
public:
   Point(double a=0, double b=0):x(a),y(b){}
   double xOffset()const{ return x; }
   double yOffset()const{ return y; }
   void moveTo(double a, double b){ x=a, y=b; }
   void print(){ cout<<"("<<x<<","<<y<<")\n"; }
}; //--------------------
class Circle{                //圆-组合类
   Point point;
   double radius;
public:
   Circle(const Point& p, double r):point(p),radius(r){}
   double getRadius()const{ return radius; }
```




```
    Point getPoint()const{ return point; }
    double getArea()const{ return radius * radius * 3.14; }
    double getCircum()const{ return 2 * radius * 3.14; }
    void moveTo(double a, double b){ point.moveTo(a, b); }
    void modifyRadius(double r){ radius = r; }
}; //--------------------
int main(){
    Point a(2.3, 5.6);
    Circle c(a, 7);
    c.moveTo(1, 2);
    c.modifyRadius(3);
    Point q = c.getPoint();
    cout <<"radius: "<< c.getRadius()<<"\n";
    cout <<"point: ";
    q.print();
    cout <<"area: "<< c.getArea()<<"\n";
    cout <<"circum: "<< c.getCircum()<<"\n";
}//---------------------
```

运行结果为：

```
radius: 3
point: (1,2)
area: 28.26
circum: 18.84
```

16.6　多继承如何工作

到目前为止，所讨论的类层次中，每个类只继承一个父辈，在现实世界中事情通常是这样的。但是一些类却代表两个类的合成。例如，两用沙发，它是一个沙发，也是一张床，两用沙发应允许同时继承沙发和床的特征，即 SleeperSofa 继承 Bed 和 Sofa 两个类，如图 16-3 所示。其程序代码如下：

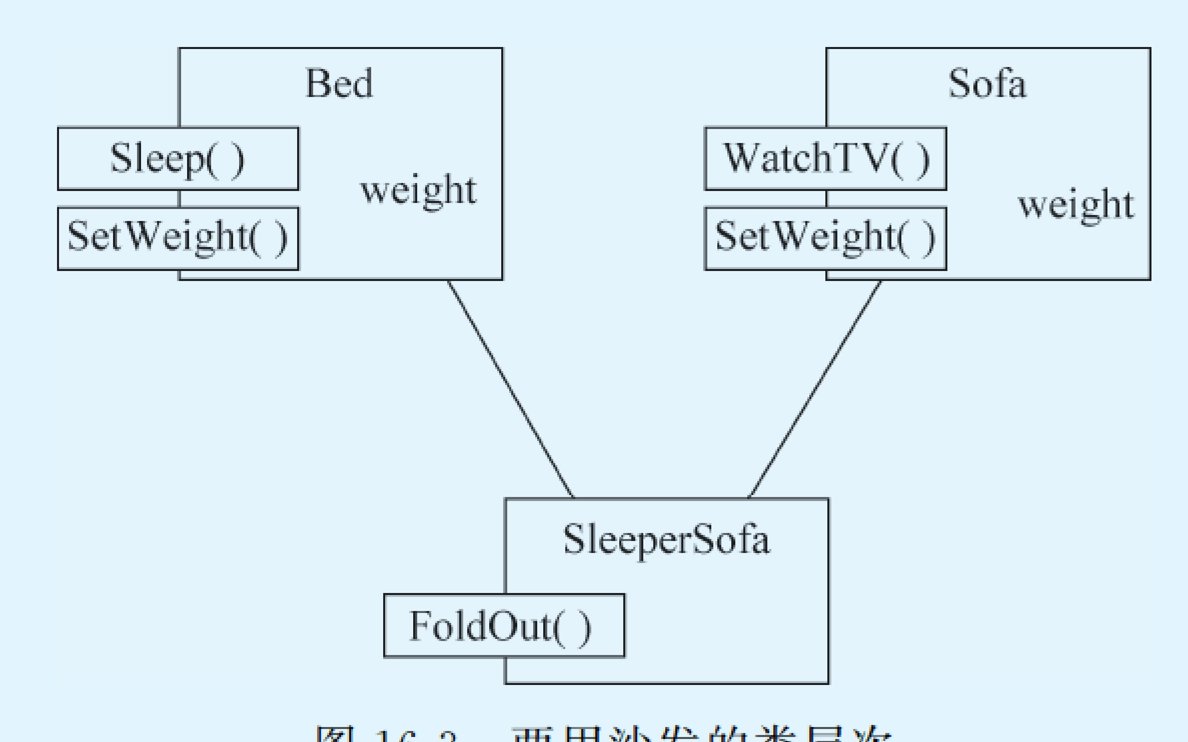


图 16-3　两用沙发的类层次

```
//---------------------
//    ch16_4.cpp
//---------------------
#include <iostream>
```

```
using namespace std;
//-----------------------
class Bed{
public:
  Bed() :weight(0){}
  void Sleep(){ cout <<"Sleeping...\n"; }
  void SetWeight(int i){ weight = i; }
protected:
  int weight;
}; //--------------------
class Sofa{
public:
  Sofa() :weight(0){}
  void WatchTV(){ cout <<"Watching TV.\n"; }
  void SetWeight(int i){ weight = i; }
protected:
  int weight;
}; //--------------------
class SleeperSofa : public Bed, public Sofa{
public:
  SleeperSofa(){}
  void FoldOut(){ cout <<"Fold out the sofa.\n"; }
}; //--------------------
int main(){
  SleeperSofa ss;
  ss.WatchTV();
  ss.FoldOut();
  ss.Sleep();
}//---------------------
```

运行结果为：

```
Watching TV.
Fold out the sofa.
Sleeping...
```

两用沙发继承两个基类的所有成员，这样 ss.Sleep()和 ss.WatchTV()的调用是合法的。也就可以把 SleeperSofa 类当作一个 Bed 类或一个 Sofa 类使用。另外，SleeperSofa 类还有它自己的成员 FoldOut()。

16.7 多继承的模糊性

在上节中，Sofa 和 Bed 都有一个成员 weight，这是合理的，因为两者都是实体，都有一个重量。问题是 SleeperSofa 继承哪个重量？既然两者都继承，由于两个父类成员有相同的名字 weight，使得对 weight 的引用变得模糊不清。

例如，按照下面引用：

```
int main()
{
   SleeperSofa ss;
   ss.SetWeight(20);      //Bed 的 SetWeight 还是 Sofa 的 SetWeight?
}
```

这样导致了名称冲突(name collision),在编译时将被拒绝。

程序必须在重量前面说明基类:

```
int main()
{
   SleeperSofa ss;
   ss.Sofa.SetWeight(20);      //说明是沙发重量 20 斤
}
```

在编写应用程序时,程序员还得额外知道类的层次信息,加大了复杂度。这些在单继承中是不会出现的。

16.8 虚拟继承

从现实意义上来看,一个 SleeperSofa 没有沙发和床两种重量,如此的继承不是真实的现实世界描述。进一步分析可得,床和沙发都是家具的一种,凡家具都有重量,所以通过分解来考察其关系,如图 16-4 所示。

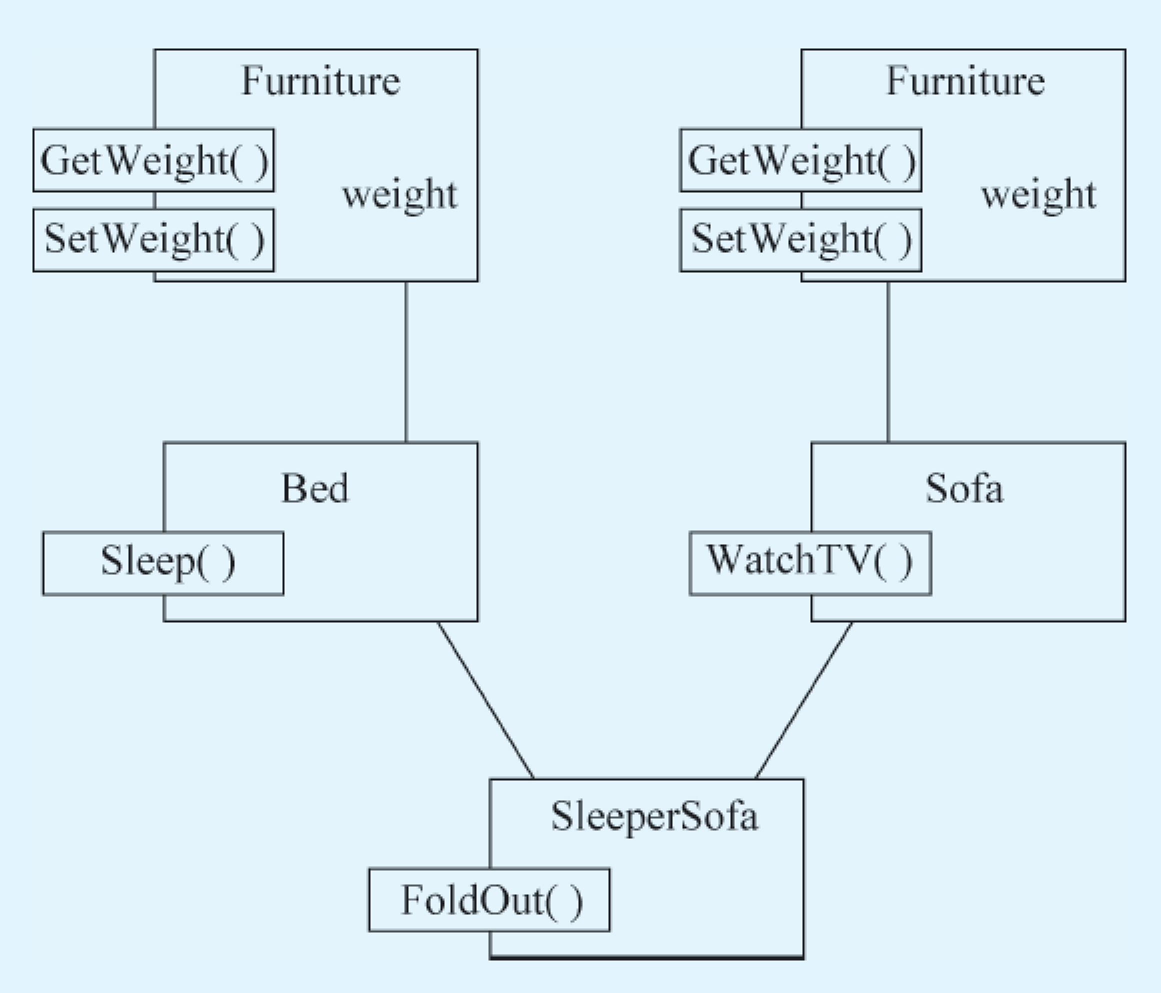


图 16-4　床和沙发的分解

```
//---------------------
//     ch16_5.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Furniture{
public:
  Furniture(){}
  void SetWeight(int i){ weight = i; }
  int GetWeight(){ return weight; }
protected:
  int weight;
}; //-------------------
```

```
class Bed : public Furniture{
public:
  Bed(){}
  void Sleep(){ cout <<"Sleeping...\n"; }
}; //--------------------
class Sofa : public Furniture{
public:
  Sofa(){}
  void WatchTV(){ cout <<"Watching TV.\n"; }
}; //--------------------
class SleeperSofa : public Bed, public Sofa{
public:
  SleeperSofa() :Sofa(), Bed(){}
  void FoldOut(){ cout <<"Fold out the sofa.\n"; }
}; //--------------------
int main(){
  SleeperSofa ss;
  ss.SetWeight(20);          //编译出错!模糊的 SetWeight 成员
  Furniture * pF;
  pF = (Furniture * )&ss;    //编译出错!模糊的 Furniture *
  cout << pF -> GetWeight()<< endl;
}//---------------------
```

因为 SleeperSofa 不是直接继承 Furniture，而是 Bed 和 Sofa 各自继承 Furniture，所以完整的 SleeperSofa 对象内存布局如图 16-5 所示。

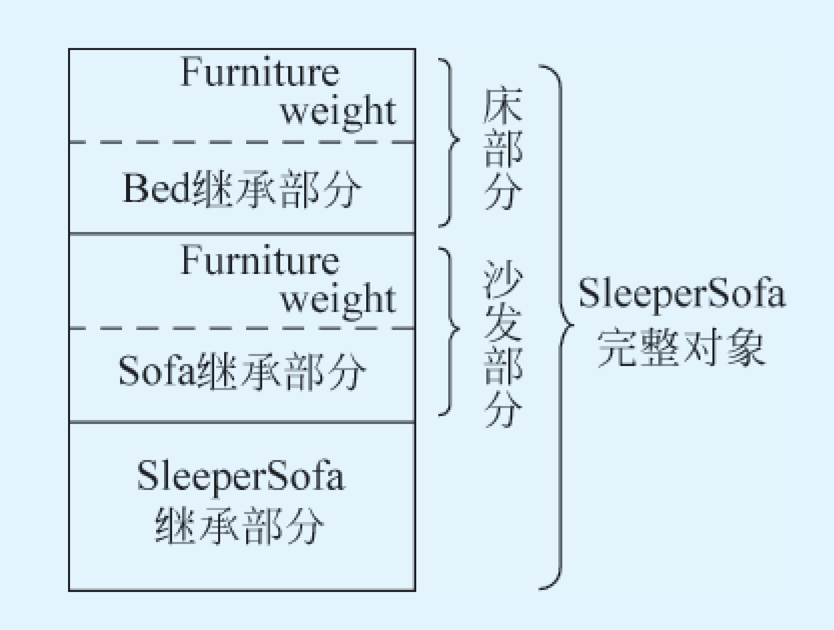


图 16-5　完整 SleeperSofa 对象内存布局

这里一个 SleeperSofa 包括一个完整的 Bed，随后还有一个完整的 Sofa，后面还有一个 SleeperSofa 特有的东西。SleeperSofa 中的每一个子对象都有它自己的 Furniture 部分。因为每个子对象都继承 Furniture，所以一个 SleeperSofa 包含两个 Furniture 对象，实际上的继承层次如图 16-6 所示。

编译 ch16_5.cpp 时，不知道 SetWeight()属于哪一个 Furniture 成员，指向 Furniture 的指针也不知道究竟指哪一个 Furniture。这就是为什么 ch16_5.cpp 编译通不过的原因。

SleeperSofa 只需一个 Furniture，所以我们希望它只含一个 Furniture 拷贝，同时又要共享 Bed 和 Sofa 的成员函数与数据成员，C++实现这种继承结构的方法称为虚拟继承(virtual inheritance)。

下面是虚拟继承的代码：



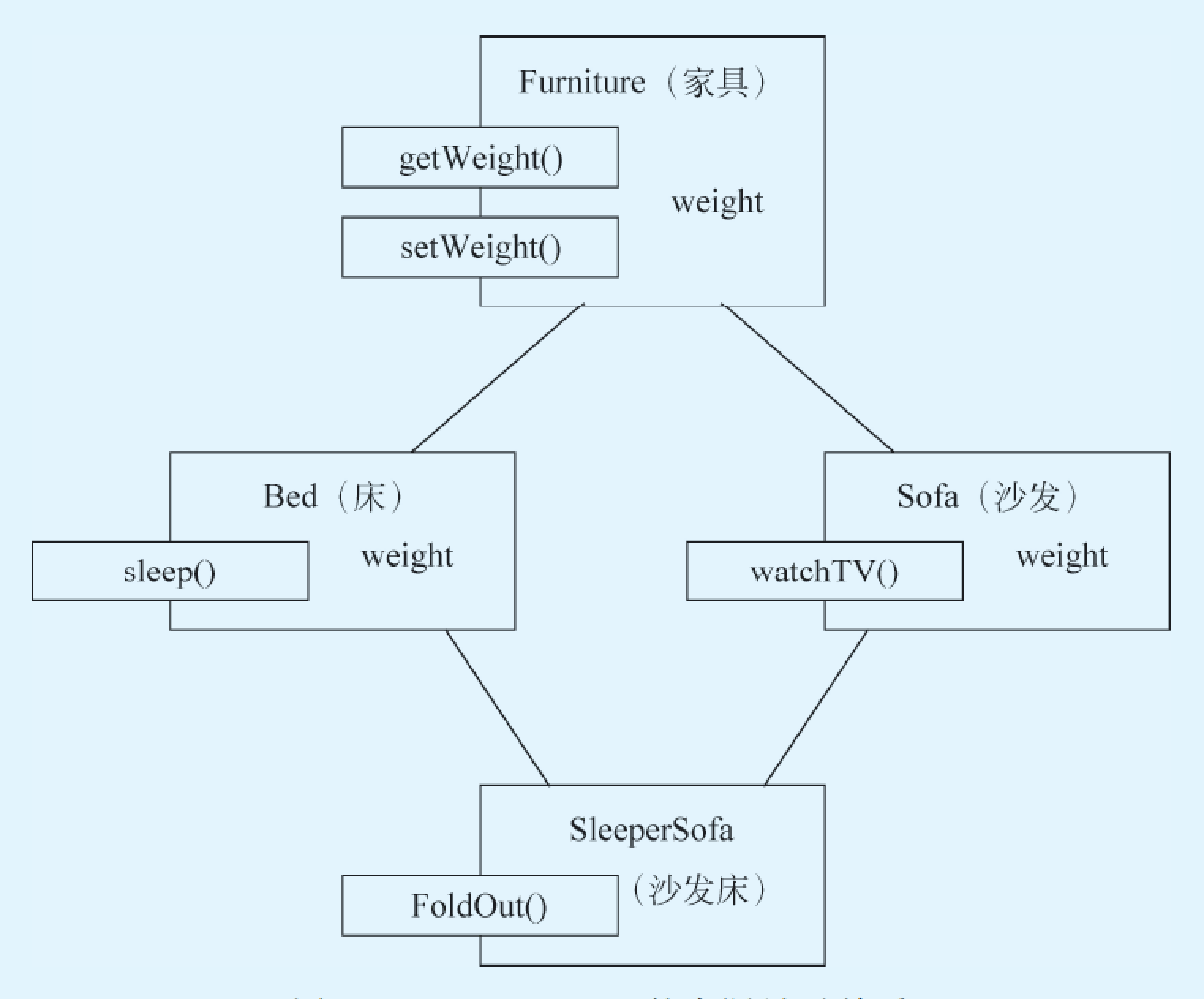


图 16-6　SleeperSofa 的实际继承关系

```
//----------------------
//     ch16_6.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
class Furniture{
public:
  Furniture(){}
  void SetWeight(int i){ weight = i; }
  int GetWeight(){ return weight; }
protected:
  int weight;
}; //--------------------
class Bed : virtual public Furniture{
public:
  Bed(){}
  void Sleep(){ cout <<"Sleeping...\n"; }
}; //--------------------
class Sofa : virtual public Furniture{
public:
  Sofa(){}
  void WatchTV(){ cout <<"Watching TV.\n"; }
}; //--------------------
class SleeperSofa : public Bed, public Sofa{
public:
  SleeperSofa() :Sofa(), Bed(){}
  void FoldOut(){ cout <<"Fold out the sofa.\n"; }
}; //--------------------
int main(){
  SleeperSofa ss;
  ss.SetWeight(20);
  cout << ss.GetWeight()<< endl;
}//--------------------
```

运行结果为：

```
20
```

在 Bed 和 Sofa 继承 Furniture 中加上 virtual 关键字，这相当于说，"如果还没有 Furniture 类，则加入一个 Furniture 拷贝，否则就用已有的那一个。"此时一个 SleeperSofa 在内存中的布局见图 16-7。

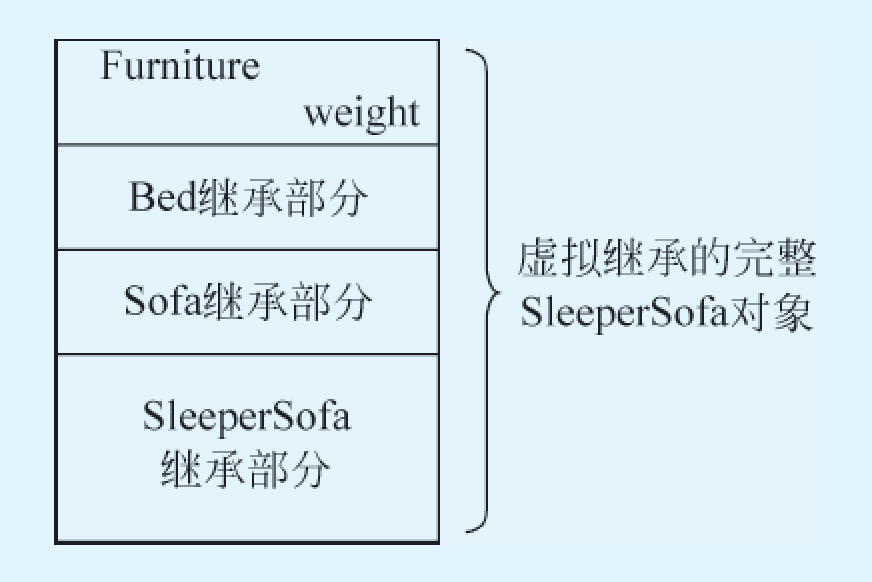


图 16-7 虚拟继承的 SleeperSofa 内存布局

在虚拟继承的情况下，应用程序 main()中引用 GetWeight()不再模糊，我们得到了真正的图 16-4 所示的继承关系。

→ *虚拟继承的虚拟和虚拟函数的虚拟没有任何关系。*

16.9 多继承的构造顺序

构造对象的规则需要扩展以控制多重继承。构造函数按下列顺序被调用：

(1) 任何虚拟基类的构造函数按照它们被继承的顺序构造；

(2) 任何非虚拟基类的构造函数按照它们被继承的顺序构造；

(3) 任何成员对象的构造函数按照它们声明的顺序调用；

(4) 类自己的构造函数。

例如：

```
//----------------------
//    ch16_7.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class OBJ1{
public:
  OBJ1(){ cout <<"OBJ1\n"; }
}; //-------------------
class OBJ2{
public:
  OBJ2(){ cout <<"OBJ2\n"; }
}; //-------------------
```




```
class B1{
public:
  B1(){ cout <<"Base1\n"; }
}; //--------------------
class B2{
public:
  B2(){ cout <<"Base2\n"; }
}; //--------------------
class B3{
public:
  B3(){ cout <<"Base3\n"; }
}; //--------------------
class B4{
public:
  B4(){ cout <<"Base4\n"; }
}; //--------------------
class Derived:public B1,virtual public B2,public B3,virtual public B4{
public:
  Derived():B4(),B3(),B2(),B1(),obj2(),obj1(){
    cout <<"Derived ok.\n";
  }
protected:
  OBJ1 obj1;
  OBJ2 obj2;
}; //--------------------
int main(){
  Derived aa;
  cout <<"This is ok.\n";
}//--------------------
```

运行结果为：

```
Base2
Base4
Base1
Base3
OBJ1
OBJ2
Derived ok.
This is ok.
```

Derived 的虚基类 Base2 和 Base4 最先构造，尽管它在 Derived 类中出现的顺序不在最前面；Derived 的非虚基类其次构造，不管它在 Derived 构造函数中出现的顺序如何；Derived 的组合对象 obj1 和 obj2 随后构造，它以类定义时数据成员排列顺序为准，不管在 Derived 构造函数中出现顺序怎样；最后是 Derived 类构造函数本身。

→ *在语言中实现多继承并不容易，这主要是编译程序问题，还有模糊性问题。如果可能，建议在进一步阅读有关参考书之前，尽量避免用多重继承。单个继承提供了足够强大的功能，不一定非用多重继承不可。我们应先学会阅读一些商品化的类库源程序中有关多重继承的部分，因为那些都是经过测试的安全代码。*

小结

C++支持类的继承机制。继承是面向对象设计的关键概念之一。有了继承,才使面向对象程序设计真正进入了实用。

派生类可以继承基类的所有公有和保护的数据成员和成员函数。

保护的访问权限对于派生类来说是公有的,而对于其他的对象来说是私有的。即使是派生类也不能访问基类中私有的数据成员和成员函数。

C++支持多重继承,从而大大增强了面向对象程序设计的表达能力。

多重继承是一个类从多个基类派生而来的能力。派生类实际上获取了所有基类的特性。当一个类是两个或多个基类的派生类时,必须在派生类名和冒号之后,列出所有基类的类名,基类之间用逗号隔开。

无论是单继承还是多继承,派生类的构造函数必须激活所有基类的构造函数,并把相应的参数传递给它们。

在面向对象的程序设计中,继承和多重继承一般指公共继承。在无继承的类中,protected 和 private 控制符是没有差别的。在继承中,基类的 private 对所有的外界都屏蔽(包括自己的派生类),基类的 protected 控制符对应用程序是屏蔽的,但对其派生类是可访问的。

在类编程中,公有继承形式占大多数,保护继承和私有继承很少。

练习

16.1 找出下列程序的错误。

```
#include<iostream>
using namespace std;
//-------------------
class A{
  int x;
public:
  A(int a):x(a){ cout<<"Constructing A\n"; }
}; //-----------------
class B : public A{
  public:
  B(){ cout<<"Constructing B\n"; }
}; //-----------------
int main(){
  B b;
}//-----------------
```

16.2 为图 16-1 设计一个类层次结构,其中的每个类都有构造函数,且有启动、停止操作。编制应用程序,创建大客车和本田小轿车,分别有启动和停止操作,输出一些标志性字串。

16.3 给定如图 16-8 所示的继承图,写出程序代码,在应用程序中,建立 C 类对象,访问 A



类中的成员函数以设置和读取其数据成员。

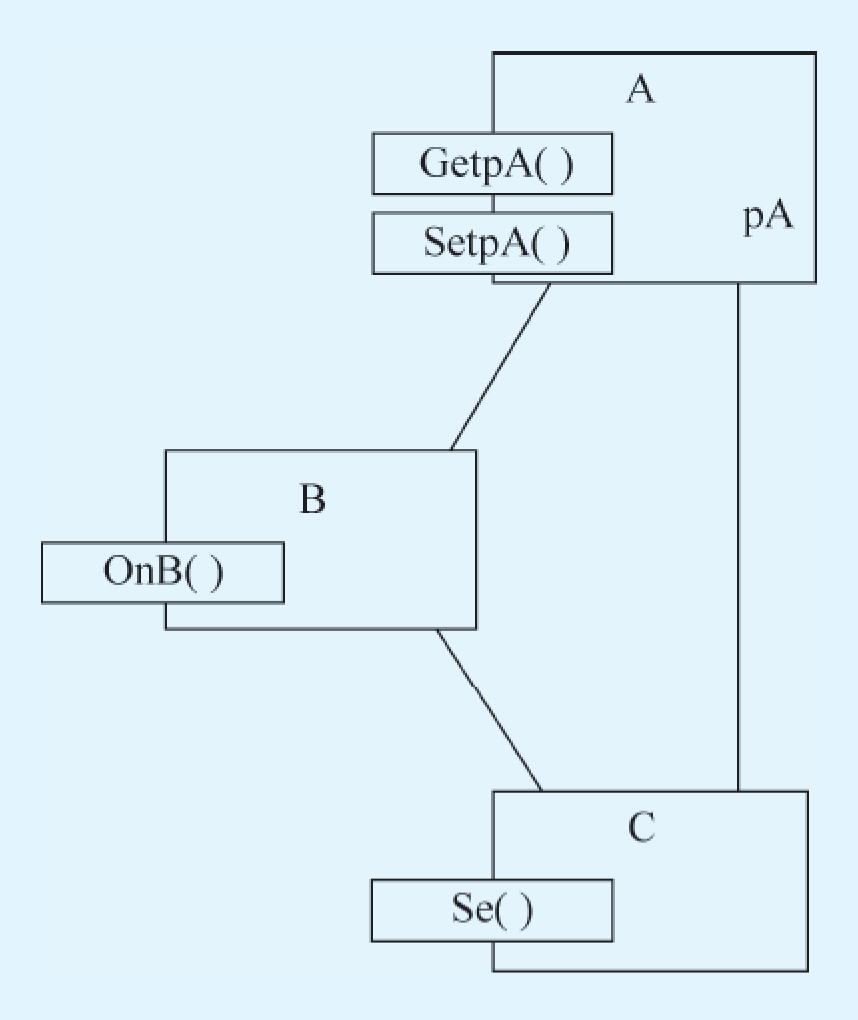


图 16-8　继承图

16.4　一个三口之家,大家都知道其父亲会开车,母亲会唱歌。但是其父亲还会修电视机,只有家里人知道。小孩既会开车又会唱歌甚至也会修电视机。母亲瞒着任何人在外面做小工以补贴家用。此外小孩还会打乒乓球。

编制程序,让这三口之家从事一天的活动:先是父亲出去开车,然后母亲出去工作(唱歌)母亲下班后去做两小时小工。小孩在俱乐部打球,在父亲回家后,开车玩,后又高兴地唱歌。晚上,小孩和父亲一起修电视机。

后来父亲的修电视机技术让大家知道了,人们经常上门要他修电视机。这时,程序要作什么样的变动?

第17章 多态

要使对象能重用，必然要用到继承机制。继承架起了 C++ 中类家族的层次结构。然而要使类家族中的对象真正融在彼此不分的大家族中，还需要多态机制。多态在继承机制中起到了画龙点睛的作用，所以也是全书的重点之一。

通过学习本章，理解多态性对于继承的意义，掌握多态的工作原理，理解真正的多态是要维护类编程的独立性，不干扰应用编程；理解抽象类和具体类的区别，特别是看到多态支持抽象类时，对抽象编程的理解，学会运用纯虚函数。

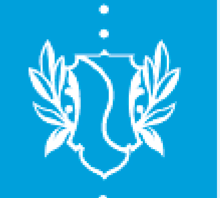
17.1 多态性

在程序 ch16_1.cpp 中，GraduateStudent 类对象 gs 调用 Student 类的成员函数 display()，该函数在输出时，没办法输出 GraduateStudent 自己的数据成员 qualifierGrade。因此在使用继承时，希望重载 display()。

C++ 允许子类的成员函数重载基类的成员函数。例如，下面的代码中，基类和派生类中都定义了计算学费的成员函数：

```
class Student
{
  public:
  //...
  float calcTuition(){      //计算学费
    //...
  }
};

class GraduateStudent :public Student
{
  public:
  //...
  float calcTuition(){
    //...
  }
```




```
};

int main()
{
  Student s;
  GraduateStudent gs;
  s.calcTuition();          //调用 Student::calcTuition()
  gs.calcTuition();         //调用 GraduateStudent::calcTuition()
}
```

派生类中重载了 calcTuition()成员函数,大学生的学费计算与研究生的学费计算方法不同。大学生对象 s 调用其学费计算函数显然指的是 Student:: calcTuition()成员,而研究生对象 gs 调用其学费计算函数时,两个重载函数都在它自己可使用范围内,C++编译规定:gs. calcTuition()指的是 GraduateStudent::calcTuition()。若派生类没有定义其 calcTuition(),则 gs. calcTuition()才指的是基类 Student::calcTuition()。

有时候,会碰到对象所属的类不能确定的情况。例如将上例作一改动:

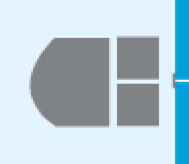
```
 class Student
 {
   public:
     //...
     float calcTuition()      //计算学费
     {
       //...
     }
};

class GraduateStudent :public Student
{
  public:
    //...
    float calcTuition()
    {
      //...
    }
};

void fn(Student& x)
{
  x.calcTuition();
}

int main()
{
  Student s;
  GraduateStudent gs;
  fn(s);                   //计算一下大学生 s 的学费
  fn(gs);                  //计算一下研究生 gs 的学费
}
```

因为大学生和研究生都是大学生,所以要求函数 fn()的形参应该能接纳这两种对象,这是继承机制的起码要求。但是接下来在函数 fn()中有一个问题:



以 fn(s)调用时,形参 x 为 Student 对象,x.calcTuition()则指 Student::calcTuition()。

以 fn(gs) 调用时,形参 x 为 GraduateStudent 对象,x.calcTuition() 应该指的是 GraguateStudent::calcTuition()。

函数调用的实际参数,随应用程序的运行进展而变更,要求函数 fn()的行为也要随着变更。这样,函数 fn()的运行执行代码就无法在编译时被确定。它一会儿调用 Student::calcTuition(),一会儿又调用 GraduateStudent::calcTuition()。

C++的继承机制中用一种称为多态性(polymorphism)的技术来解决上面的问题。这种在运行时,能依据其类型确认调用哪个函数的能力,称为多态性,或称迟后联编(late binding),也有的译为滞后联编。

编译时就能确定哪个重载函数被调用的,称为先期联编(early binding)。本节中前一个例子是先期联编的,后一个例子是迟后联编的。

17.2 多态的思考方式

若语言不支持多态,则不能被称为面向对象的语言。只支持类而不支持多态,只能称其为基于对象的语言。

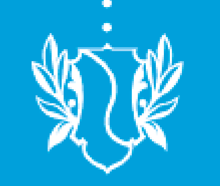
上节的函数由于增加了研究生学费计算而使学费计算操作 calcTuition()呈多态。假如处理该函数的计算机无法分辨大学生与研究生学费计算的不同,那么只有人为增加一段操作代码来弥补。

在 Student 类中增加一个 type 公共数据成员(以使普通函数能够访问它),标识大学生和研究生,分别修改 Student 和 GraduateStudent 的构造函数,对 type 成员赋初值,并在应用程序中,增加学生类别的判断,以便调用相应的学费计算操作:

```
enum StudentType{STUDENT, GRADUATESTUDENT};

class Student
{
  public:
    Student(char * pName = "no name")
    {
      strncpy(name, pName);
      average = semesterHours = 0;
      type = STUDENT;        //大学生类,标识 STUDENT
    }
    float calcTuition();
    StudentType type;        //说明为公共的,便于应用程序访问
  //...
};

class GraduateStudent :public Student
{
  public:
    GraduateStudent()
    {
```




```
        type = GRADUATESTUDENT;        //覆盖刚赋的 STUDENT 值
      }
      float calcTuition();
    //...
  };

  void fn(Student& s)
  {
    //...
    switch(s.type)                     //判断对象的 type,以确定调用哪个学费计算成员
    {
      case STUDENT:
        s.Student::calcTuition(); break;
      case GRADUATESTUDENT:
        staTic_cast<GraduateStudent>(s).calcTuition(); break;
    }
    //...
  }
```

像这种多态的成员也许不止一个,都要进行类似的处理。

何况还会继承新的类,例如继承一个博士生类,这样,应用程序的维护量很大,类的内部实现和应用程序都不能避免修改,面向对象的优越性被遏制,又回到面向过程程序设计的老路上去了。因此,避免上述程序代码,实现多态技术,是面向对象程序设计的关键之一。

计算学费的学生管理员好像一个"迟后联编"者,别人关心的是他工作的内容,即计算所有学生的学费,并不关心计算方法:如果是大学生,则按一计算公式计算学费;如果是研究生,则按另一计算公式计算学费,等等。具体的操作(各个成员函数)决定于学生管理员自己,该用什么计算公式取决于当时实际的学生性质。

对于学生管理员,这样的工作方式体现了其工作效率,他要比一个不知道区分学生性质的新手要强。另一方面,只要不是做同一件事,人与人之间的思想交流,一般总是在更高的抽象层面上,这样有利于更直接和精确地把握思考的事物。例如,一个人对另一个人吆喝"开门",彼此都处于同一场景中,明白是开机舱门还是开冰箱门等等。这就是人的思考问题的方式,也就是自然的多态方式。沿着这种思路设计的程序,可使软件模型更准确地描述人们所思考的问题。

17.3 多态性如何工作

为了指明某个成员函数具有多态性,用关键字 virtual 来标志其为虚函数。例如:

```
//----------------------
//    ch17_1.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
```

```
class Base{
public:
  virtual void fn(){              //基类中的虚函数
    cout <<"In Base class\n";
  }
}; //--------------------
class SubClass : public Base{
public:
  virtual void fn(){
    cout <<"In SubClass\n";     //子类中的虚函数
  }
}; //--------------------
void test(Base& b){
  b.fn();
}//---------------------
int main(){
  Base bc;
  SubClass sc;
  cout <<"Calling test(bc)\n";
  test(bc);
  cout <<"Calling test(sc)\n";
  test(sc);
}//---------------------
```

运行结果为：

```
Calling test(bc)
In Base Class
Calling test(sc)
In SubClass
```

fn()是 Base 类的虚函数，在 test()函数中，b 是基类 Base 的传引用形参，Base 类对象和 SubClass 类对象都可作为参数传递给 b，所以 b. fn()的调用要等到运行时，才能确认是调用基类的 fn()还是子类的 fn()。

由于 fn()标志为虚函数，编译看见 b. fn()后，将其作为迟后联编来处理，以保证在运行时确定调用哪个 fn()虚函数。

编译通常是在先期联编状态下工作的，只有看见虚函数，才把它作为迟后联编来实现。多态性增加了一些数据存储和执行指令的代价，但与所得到的编程灵活性和方便性相比，还是值得的。

fn()在基类中声明为 virtual，该虚函数的性质自动地向下带给其子类，所以 SubClass 子类中 virtual 可以省略。

例如，下面的程序进一步说明多态的使用。圆类和长方形类分别继承于形状 Shape 类，由一个函数专门负责求某圆或某长方形对象的面积：

```
//----------------------
//    ch17_2.cpp
//----------------------
#include <iostream>
```




```
#include <cmath> //用到 abs()
using namespace std;
//----------------------
class Shape{
public:
  Shape(double x, double y) : xCoord(x), yCoord(y){}
  virtual double Area()const{ return 0.0; }
protected:
  double xCoord, yCoord;
}; //--------------------
class Circle : public Shape{
public:
  Circle(double x, double y, double r) : Shape(x, y), radius(r){}
  virtual double Area()const{ return 3.14 * radius * radius; }
protected:
  double radius;
}; //--------------------
class Rectangle : public Shape{
public:
  Rectangle(double x1, double y1, double x2, double y2)
         : Shape(x1,y1), x2Coord(x2), y2Coord(y2){}
  virtual double Area()const;
protected:
  double x2Coord, y2Coord;
}; //--------------------
double Rectangle::Area()const{    //虚函数在类外定义
  return abs((xCoord - x2Coord) * (yCoord - y2Coord));
}//---------------------
void fun(const Shape& sp){
  cout << sp.Area()<< endl;
}//---------------------
int main(){
  Circle c(2.0,5.0,4.0);
  fun(c);
  Rectangle t(2.0,4.0,1.0,2.0);
  fun(t);
}//---------------------
```

运行结果为:

```
50.24
2
```

fun()函数负责计算所有对象的面积,它知道只要调用求面积的 Area()函数就行了,不管参数是什么类型的对象,C++的迟后联编会做好这一切的。这样一来,fun()省了很多心,程序显得简单,以后要增加一个求新的形状的面积,只要简单地派生一个类就行了,应用程序不用作修改。

多态性让类的设计者更多地去考虑工作的细节,而且这个细节简单,就是在成员函数前加一个 virtual 关键字。多态性使应用程序代码极大地简化,它是开启继承能力的钥匙。

17.4 不恰当的虚函数

1. 未出自基类

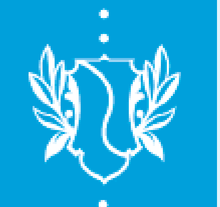
如果虚函数在基类与子类中仅仅是名字相同,但参数类型不同,或返回类型不同,这样即使写上了 virtual 关键字,系统也不进行迟后联编。

例如,下面程序在派生类中重载了基类的成员函数,尽管函数标上了 virtual,运行中还是起不到多态的效果:

```
//---------------------
//    ch17_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Base{
public:
  virtual void fn(int x){
    cout << "In Base class, int x = " << x << endl; }
}; //-------------------
class SubClass : public Base{
public:
  virtual void fn(float x){
    cout << "In SubClass, float x = " << x << endl;
  }
}; //-------------------
void test(Base& b){
  int i = 1;
  b.fn(i);
  float f = 2.0;
  b.fn(f);
}//---------------------
int main(){
  Base bc;
  SubClass sc;
  cout << "Calling test(bc)\n";
  test(bc);
  cout << "Calling test(sc)\n";
  test(sc);
}//---------------------
```

运行结果为:

```
Calling test(bc)
In Base class, int x = 1
In Base class, int x = 2
Calling test(sc)
In Base class, int x = 1
In Base Class, int x = 2
```



基类中的 void fn(int x)和子类中的 void fn(float x)是两个不同的函数,它们只是同名函数重载。SubClass 类继承了 Base 类,但是 SubClass 类中却没有实现多态的 voi fn(int x)函数。Subclass 又自己添了一个多态成员函数 void fn(float x),但只能对 SubClass 派生的子类产生作用。看清这些,对 test (Base& b)函数中的 b.fn(i)成员函数调用的理解就十分简单。

对于传自 Base 类对象的引用 b,编译看见 b.fn(i)的调用,分析到 fn(i)只有在 Base 类中定义的一个版本,无论是基类对象还是子类对象,调用的都是 Base::fn(int)成员,不存在多态,也就无须迟后联编了。即使后面调用 b.fn(f),对于 Base 类唯一的函数,也只能对 float 作 int 转换而去匹配 void fn(int)。

而对于传自 SubClass 类对象的引用 b,编译看见 b.fn(i)的调用,分析到 fn(i)虽然是多态函数,但是 SubClass 却没有自己的版本,对于首要的精准匹配性,自然只能调用 Base::fn(int)了,所以也就无须迟后联编了。

接下来编译又看见 b.fn(f)的调用。首先应该明确,子类对象 b 是作为 Base 基类对象被匹配的,除非调用的是基类传播下来的多态函数,否则匹配的唯有基类成员函数。b.fn(f)只在 Base 类中寻求匹配,哪怕委屈地将 float 转换成 int。编译绝不会去尝试寻求和子类的 fn(float)匹配。

2. 可返回不同类

上述程序的编译分析与运行结果,关键是因为不恰当的虚函数没有构成多态,使编译看作为先期联编。有一种例外,如果基类中的虚函数返回一个基类指针或返回一个基类的引用,子类中的虚函数返回一个子类的指针或子类的引用,则 C++将其视为同名虚函数并进行迟后联编。

例如,下面的程序中,派生类与基类的成员函数返回类型不同,但仍起到虚函数的作用:

```
//----------------------
//    ch17_4.cpp
//----------------------
#include<iostream>
using namespace std;
//----------------------
class Base{
public:
  virtual Base* afn(){
    cout<<"This is Base class.\n";
    return this;
  }
};//--------------------
class SubClass : public Base{
public:
  SubClass* afn(){
    cout<<"This is SubClass.\n";
    return this;
  }
};//--------------------
```

```
void test(Base& x){
  Base * b;
  b = x.afn();
}//--------------------
int main(){
  Base bc;
  SubClass sc;
  test(bc);
  test(sc);
}//--------------------
```

运行结果为：

```
This is Base class
This is SubClass
```

两个函数参数形式相同，返回类型不同，在调用时将体现不出差异。编译认为 Base * afn()和 SubClass * afn()是多态的，所以在看见 x.afn()的调用时进行了迟后联编。运行时视 x 对象的类型而确定调用哪个 afn()虚函数。

这个例外也是合理的，如果一个函数正处理 SubClass 的对象，则它仍可继续处理 SubClass 对象，这是很自然的事。

17.5 虚函数的限制

一个类中将所有的成员函数都尽可能地设置为虚函数对编程固然方便，Java 语言中正是这样做的，但是会增加一些时空上的开销。C++是在性能上有偏激追求的编程语言，只选择设置个别成员函数为虚函数。设置虚函数，须注意下列事项：

(1) 只有类的成员函数才能说明为虚函数。这是因为虚函数仅适用于有继承关系的类对象，所以普通函数不能说明为虚函数。

(2) 静态成员函数不能是虚函数，因为静态成员函数不受限于某个对象。例如，如果下列 staticfn()是静态成员函数：

```
void fn(Base& x)
{
  x.staticfn();          //只是用了 x 的类型信息,x 并不求值
  Base::staticfn();      //调用静态成员的推荐方法
}
```

(3) 内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时，仍将其看作非内联的。

(4) 构造函数不能是虚函数，因为构造时，对象还是一片未定型的空间。只有在构造完成后，对象才能成为一个类的名副其实的实例。

(5) 析构函数可以是虚函数，而且通常声明为虚函数。例如，当基类对象和子类对象以不同方式申请了堆空间后：

```
void finishWithObject(Base * pHeapObject)
{
  //...
```

```
        delete pHeapObject;
    }
```

pHeapObject 是传递过来的一个对象指针，它或者指向基类对象，或者指向子类对象。在执行 delete pHeapObject 时，要调用析构函数，但是执行基类的析构函数？还是执行子类的析构函数？将析构函数声明为虚的，就可以解决这个问题。

17.6 继承设计问题

1. 类的冗余

继承给程序员在一个过程中从不同类里组合共有特征的能力，这个过程称为分解(factoring)。也就是把共同的特征分解到基类中。

用分解银行存款的例子来说明，见图 17-1。

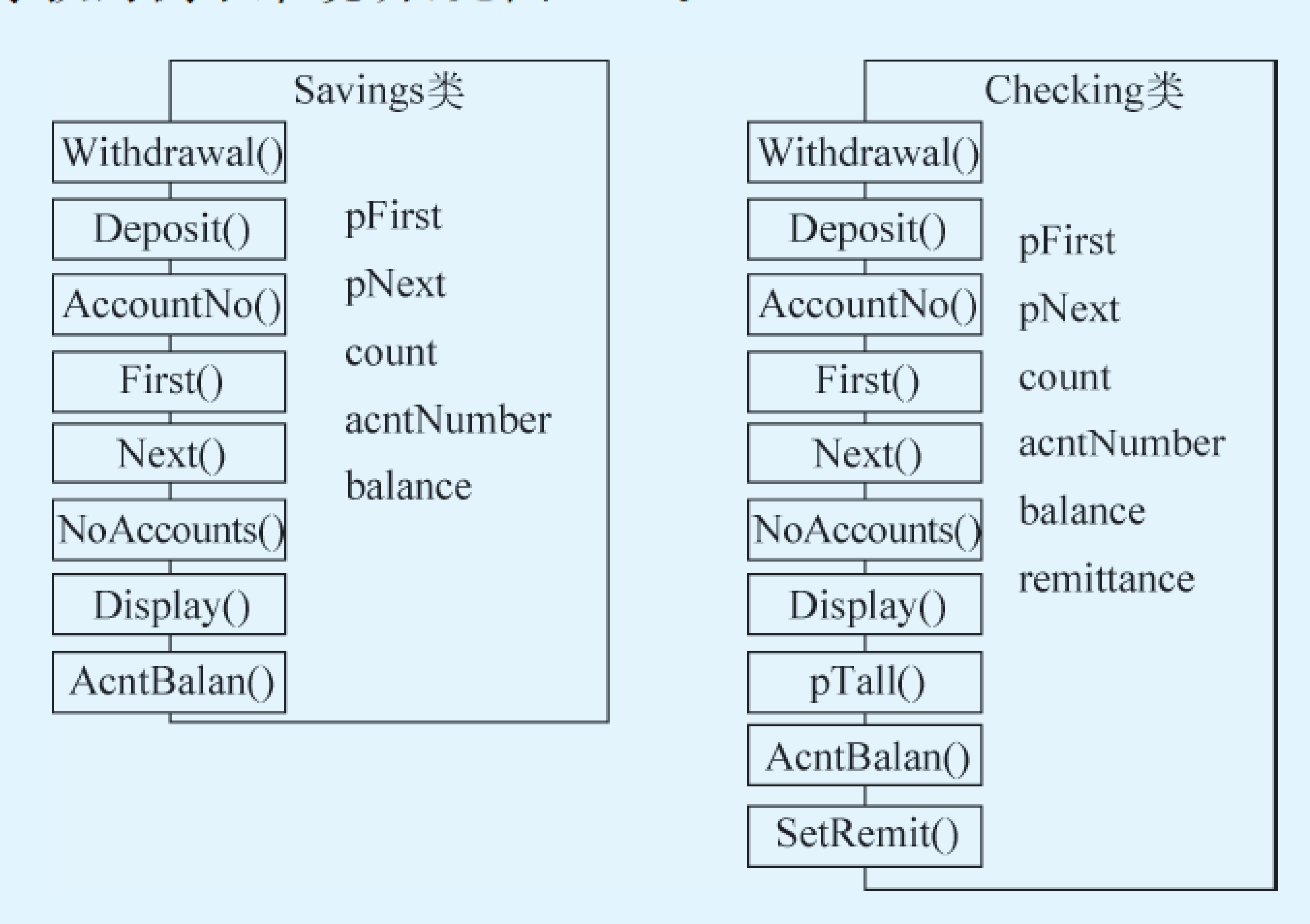


图 17-1 两个独立的银行存款类

图中有两个银行存款类，分别为 Savings(储蓄)类和 Checking(结算)类。该银行存款类是示意性的，实际在银行中的存款业务要远多于示例的内容。大方框代表类，类名在方框的顶部，小方框里面的名字是作为接口的公共成员函数，不在小方框里的名字是保护的数据成员。

所有的储蓄构成一个链表，所有的结算也构成独立的一个链表。链表中的结点个数即账户数(count)，链表首指针为 pFirst，结点中指向下一结点的指针为 pNext。银行存款一般有账号(acntNumber)和结算余额(balance)。对于结算类，还要有异地汇款的方式(remittance)、或信汇、或电汇、或其他转账结算方式。

对银行存款类的操作有 Deposit()(存款)、Withdrawal()(取款)、AccountNo()(开银行账号)，AcntBalance()(查询账户中的余额)。

为了维护银行存款类中的链表，需要有操作：First()(取链表首指针)、Next()(取下一个结点)、NoAccounts()(取链表中的账户数)。

与储蓄存款不同，对于结算存款，取款若为信汇(邮寄汇款)或电汇(电报汇款)，则要加收一定的手续费。为此，设立数据成员 remittance(汇款方式)和成员函数 SetRemit()(设置

汇款方式)，并且取款操作 Withdrawal()也与储蓄存款不同。

Savings 类和 Checking 类定义的代码分别为：

```
//---------------------
// savings.h
//---------------------
#ifndef SAVINGS
#define SAVINGS
//---------------------
class Savings{
public:
  Savings(unsigned acntNo, float balan = 0.0);
  unsigned AccountNo(){ return acntNumber; }
  float AcntBalan(){ return balance; }
  static Savings* First();                //取链首指针
  Savings* Next();                        //取结点指针
  static int NoAccounts();
  void Display()const;
  void Deposit(float amount){ balance += amount; }
  void Withdrawal(float amount);
protected:
  static Savings* pFirst;
  static Savings* pTail;
  Savings* pNext;
  static int count;
  unsigned acntNumber;
  float balance;
}; //-----------------------------------
#endif // SAVINGS
//---------------------
// savings.cpp
//---------------------
#include"savings.h"
#include<iostream>
using namespace std;
//---------------------
Savings* Savings::pFirst = 0;              //链表为空
int Savings::count = 0;                    //账户个数为 0
//---------------------
Savings::Savings(unsigned acntNo, float balan)
  :acntNumber(acntNo), balance(balan){
  count++;
  if(pFirst == 0)
    pFirst = this;
  else
    pTail->pNext = this;
  pTail = this;
  pNext = 0;
}//--------------------
void Savings::Display()const{
  cout <<"Savings Account:"<< acntNumber <<" = "<< balance <<"\n";
}//--------------------
void Savings::Withdrawal(float amount){
```




```
    if(balance < amount)
      cout <<"Insufficient funds withdrawal: "<< amount <<"\n";
    else
      balance -= amount;
}//--------------------
Savings* Savings::First(){                //取链首指针
  return pFirst;
}//--------------------
Savings* Savings::Next(){                 //取结点指针
  return pNext;
}//--------------------
int Savings::NoAccounts(){
  return count;
}//--------------------
//---------------------
// checking.h
//---------------------
#ifndef CHECKING
#define CHECKING
//---------------------
enum REMIT{remitByPost, remitByCable, other}; //信汇,电汇,无
//---------------------
class Checking{
protected;
  static Checking* pFirst;
  Checking* pNext;
  static int count;
  unsigned acntNumber;
  float balance;
  REMIT remittance;
public:
  Checking(unsigned acntNo, float balan = 0.0);
  unsigned AccountNo(){ return acntNumber; }
  float AcntBalan(){ return balance; }
  static Checking* First();               //取链首指针
  Checking* Next();                       //取结点指针
  static int NoAccounts();
  void Display()const;
  void Deposit(float amount){ balance += amount; }
  void Withdrawal(float amount);
  void setRemit(REMIT re){ remittance = re; }
}; //--------------------
#endif // CHECKING
//---------------------
// checking.cpp
//---------------------
#include"checking.h"
#include<iostream>
using namespace std;
//---------------------
Checking* Checking::pFirst = 0;           //链表为空
int Checking::count = 0;                  //账户个数为 0
//---------------------
Checking::Checking(unsigned acntNo, float balan)
  :acntNumber(acntNo), balance(balan), remittance(other){
  count++;
  if(pFirst == 0)
    pFirst = this;
```

```
    else
      pTail->pNext=this;
    pTail=this;
    pNext=0;
}//--------------------
void Checking::Display()const{
    cout<<"Checking Account:"<<acntNumber<<"  =  "<<balance<<"\n";
}//--------------------
void Checking::Withdrawal(float amount){
    if(remittance==remitByPost)          //信汇加收 30 元手续费
      amount+=30;
    if(remittance==remitByCable)         //电汇加收 60 元手续费
      amount+=60;
    if(balance<amount)
      cout<<"Insufficient funds withdrawal: "<<amount<<"\n";
    else
      balance-=amount;
}//--------------------
Checking* Checking::First(){             //取链首指针
    return pFirst;
}//--------------------
Checking* Checking::Next(){              //取结点指针
    return pNext;
}//--------------------
int Checking::NoAccounts(){
    return count;
}//--------------------
```

2. 克服冗余带来的问题

针对上述中的问题,我们让其中的一个类继承另一个类。Checking 有额外的数据成员,因此让它继承 Savings 类更合理些。如图 17-2 所示,该类通过增加数据成员 remittance 和成员函数 SetRemit()以及将成员函数 Withdrawal()设为虚函数的办法来完成。

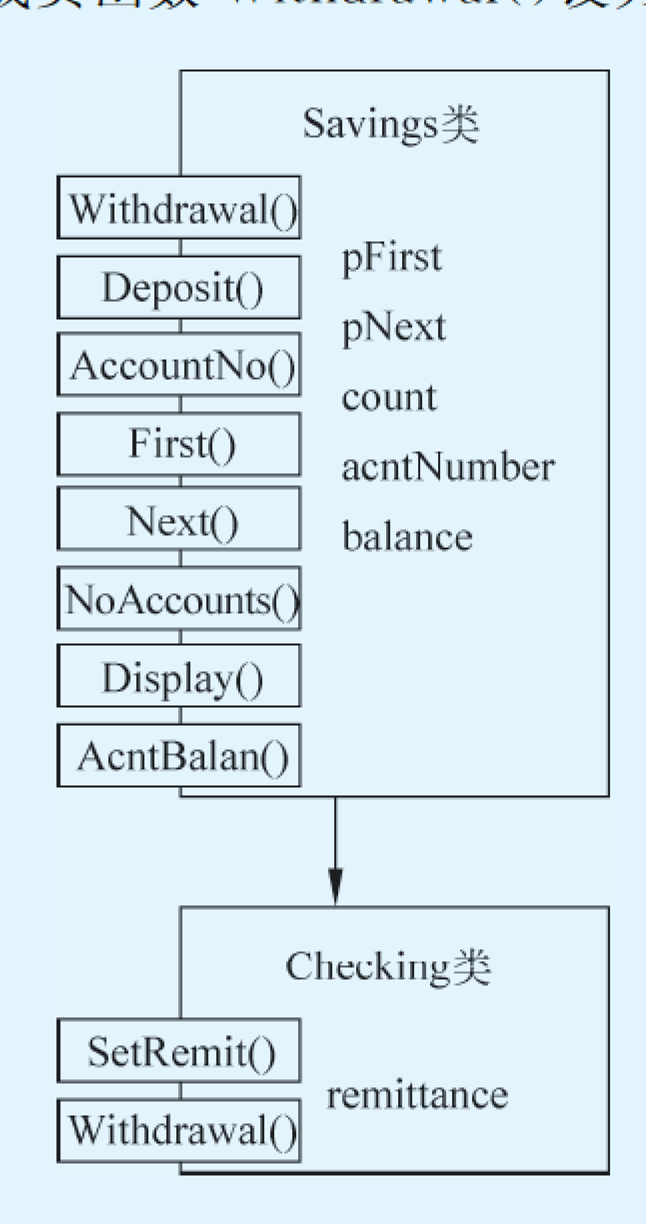
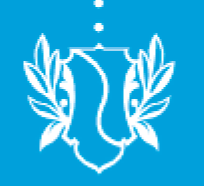


图 17-2 Checking 类作为 Savings 类的一个派生



如此一来,节省了打字的时间,Savings 和 Checking 类的界面实现可以如下设计:

```
//----------------------
// savings.h
//----------------------
#ifndef SAVINGS
#define SAVINGS
//----------------------
class Savings{
public:
  Savings(unsigned acntNo, float balan = 0.0);
  unsigned AccountNo(){ return acntNumber; }
  float AcntBalan(){ return balance; }
  static Savings* First();              //取链首指针
  Savings* Next();                      //取结点指针
  static int NoAccounts();
  void Display()const;
  void Deposit(float amount){ balance += amount; }
  virtual void Withdrawal(float amount);   //虚函数
protected:
  static Savings* pFirst;
  static Savings* pTail;
  Savings* pNext;
  static int count;
  unsigned acntNumber;
  float balance;
}; //--------------------
#endif // SAVINGS
//----------------------
// checking.h
//----------------------
#ifndef CHECKING
#define CHECKING
//----------------------
#include"savings.h"
//----------------------
enum REMIT{remitByPost, remitByCable, other}; //信汇, 电汇, 无
//----------------------
class Checking : public Savings{
protected;
  REMIT remittance;
public:
  Checking(unsigned acntNo, float balan = 0.0);
  void Withdrawal(float amount);
  void setRemit(REMIT re){ remittance = re; }
}; //--------------------
#endif // CHECKING
```

Savings 类的内部实现不必修改,而 Checking 类的内部实现则简化为:

```
//----------------------
// checking.cpp
//----------------------
#include"checking.h"
//----------------------
Checking::Checking(unsigned acntNo, float balan)          //基类初始化完成链表操作
  : Savings(acntNo, balan), remittance(other){}
```

```
//---------------------
void Checking::Withdrawal(float amount){
  float tmp = amount;
  if(remittance == remitByPost)         //信汇加收 30 元手续费
    tmp = amount + 30;
  if(remittance == remitByCable)        //电汇加收 60 元手续费
    tmp = amount + 60;
  Savings::Withdrawal(tmp);             //调用基类取款操作
}//---------------------
```

从中看出,Checking 类只包含了与 Savings 类之间不同的部分。

尽管这样的解决可以省力,但不能令人完全满意。Checking 账户继承 Savings 账户的关系,暗示一个结算账户是储蓄账户的一个特殊类型,但事实上不是。

例如,银行改变了在储蓄账户上的政策。假设银行决定储蓄账户的余额可以出现允许范围的负数(一定程度的透支)。根据这样的假定,在 Savings 账户中增加一个新数据成员 minbalance(透支范围),然后修改一下虚成员函数 Withdrawal()即可完成。

但是,问题来了。因为 Checking 类继承 Savings 类,Checking 类也得到了 minbalance 这个新数据成员。minbalance 对 Checking 类账户没用,一个额外的数据成员虽不算大,但增加了混淆。

这样的变化是日积月累的。一旦确定了类的层次关系,也就确定了系统的实现。今天增加一个额外的数据成员,明天又可能要改变成员函数。

这样的分类和继承关系能够正常工作,并且省力,这是事实。但是,类的层次必须体现其意义,否则会使程序员混淆。因为终究会有一天,一个不太熟悉该程序的程序员接手这一系统,将不得不阅读理解这些代码到底干什么,错误导向的技巧将使程序更难理解和一致化。

3. 类的分解

为了避免上节中的问题,可为 Savings 类和 Checking 类专门建立一个新类,并让这两个类都基于新类之上。不妨将该新类称之为 Account 类,该类包含了储蓄类和结算类所共有的特征,见图 17-3。

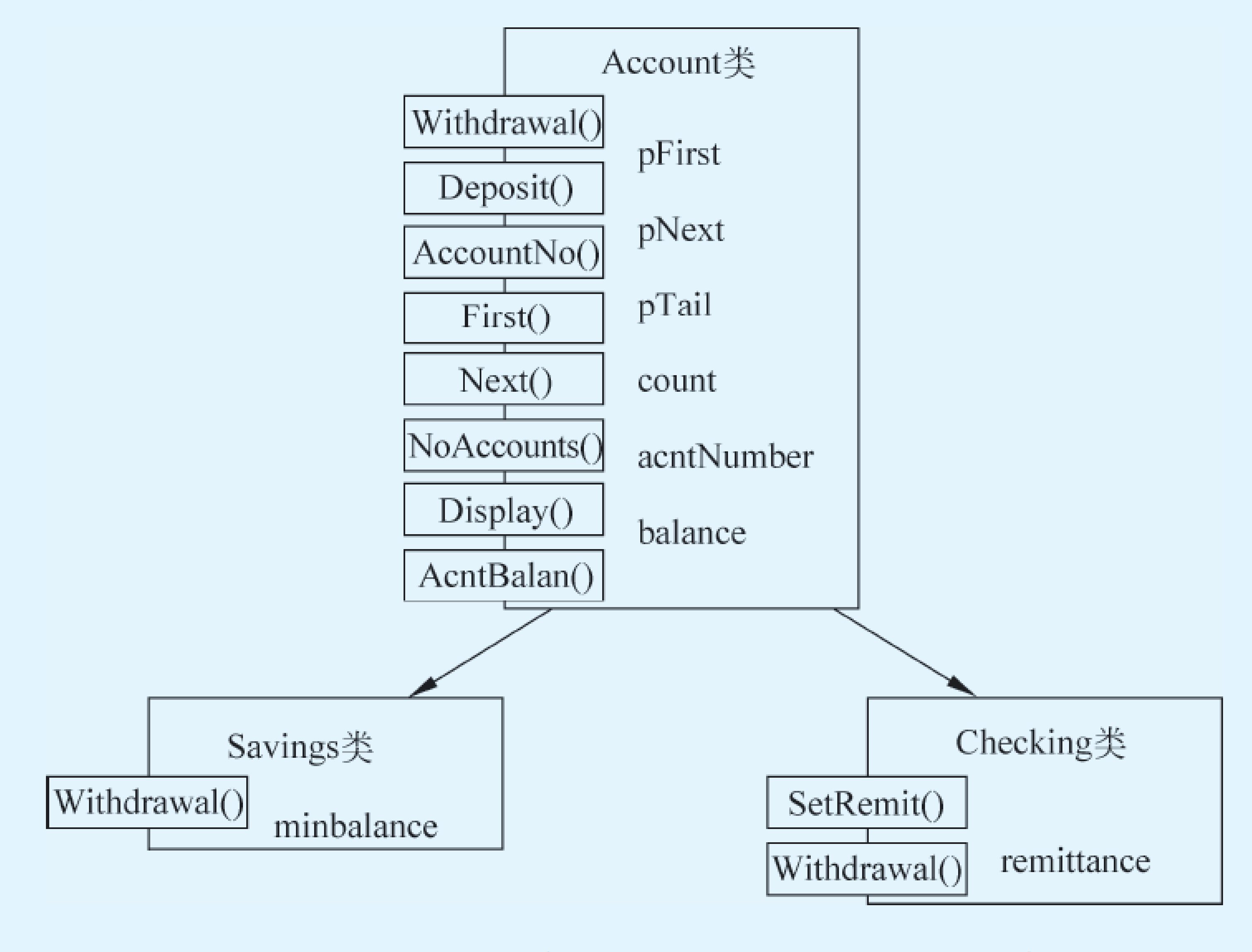
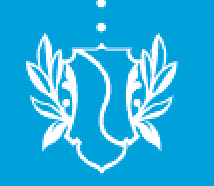


图 17-3 Account 类派生 Savings 和 Checking 类

在图中,Savings 类和 Checking 类的成员函数 Withdrawal()为虚函数。Savings 类继承了 Account 类,自身补充了 minbalance 数据成员。Checking 类继承了 Account 类,自身补充了 remittance 数据成员和 SetRemit()成员函数。相应的类代码描述为:

```
//---------------------
// account.h
//---------------------
#ifndef ACCOUNT
#define ACCOUNT
//---------------------
class Account{
public:
  Account(unsigned acntNo, float balan = 0.0);
  unsigned AccountNo(){ return acntNumber; }
  float AcntBalan(){ return balance; }
  static Account * First();              //取链首指针
  Account * Next();                      //取结点指针
  static int NoAccounts();
  void Display()const;
  void Deposit(float amount){ balance += amount; }
  virtual void Withdrawal(float amount);  //虚函数
protected:
  static Account * pFirst;
  static Account * pTail;
  Account * pNext;
  static int count;
  unsigned acntNumber;
  float balance;
}; //-----------------------------------
#endif // ACCOUNT
//---------------------
// account.cpp
//---------------------
#include"account.h"
#include<iostream>
using namespace std;
//---------------------
Account * Account::pFirst = 0;            //链表为空
int Account::count = 0;                   //账户个数为 0
//---------------------
Account::Account(unsigned acntNo, float balan)
  :acntNumber(acntNo), balance(balan){
  count++;
  if(pFirst == 0)
    pFirst = this;
  else
    pTail->pNext = this;
  pTail = this;
  pNext = 0;
}//--------------------
void Account::Display()const{
  cout <<"Account number:"<< acntNumber <<" = "<< balance <<"\n";
}//--------------------
```

```
void Account::Withdrawal(float amount){
  return; //不做任何事
}//---------------------
Account * Account::First(){            //取链首指针
  return pFirst;
}//---------------------
Account * Account::Next(){             //取结点指针
  return pNext;
}//---------------------
int Account::NoAccounts(){
  return count;
}//---------------------
//----------------------
// savings.h
//----------------------
#ifndef SAVINGS
#define SAVINGS
//----------------------
#include"account.h"
//----------------------
class Savings : public Account{
public:
  Savings(unsigned acntNo, float balan = 0.0);
  virtual void Withdrawal(float amount); //虚函数
protected:
  static float minbalance;
}; //-----------------------------------
#endif // SAVINGS
//----------------------
// savings.cpp
//----------------------
#include"savingsM.h"
#include"account.h"
#include<iostream>
using namespace std;
//----------------------
float Savings::minbalance = 500;        //设置透支范围
//----------------------
Savings::Savings(unsigned acntNo, float balan)
  : Account(acntNo, balan){}
//----------------------
void Savings::Withdrawal(float amount){
  if(balance + minbalance < amount)
    cout <<"Insufficient funds: balance "<< balance <<",withdrawal"<< amount <<"\n";
  else
    balance -= amount;
}//---------------------
//----------------------
// checking.h
//----------------------
#ifndef CHECKING
#define CHECKING
//----------------------
#include"account.h"
```




```
//----------------------
enum REMIT{remitByPost, remitByCable, other};        //信汇, 电汇, 无
//----------------------
class Checking : public Account{
protected:
  REMIT remittance;
public:
  Checking(unsigned acntNo, float balan = 0.0);
  void Withdrawal(float amount);
  void setRemit(REMIT re){ remittance  =  re; }
}; //--------------------
#endif // CHECKING
//----------------------
// checking.cpp
//----------------------
#include"checking.h"
#include"account.h"
#include<iostream>
using namespace std;
//----------------------
Checking::Checking(unsigned acntNo, float balan)     //基类初始化完成链表操作
  : Account(acntNo, balan), remittance(other){}
//----------------------
void Checking::Withdrawal(float amount){
  float tmp = amount;
  if(remittance == remitByPost)                      //信汇加收 30 元手续费
    tmp = amount + 30;
  if(remittance == remitByCable)                     //电汇加收 60 元手续费
    tmp = amount + 60;
  if(balance < tmp)
    cout <<"Insufficient funds:balance "<< balance <<", withdrawal"<< tmp <<"\n";
  else
    balance -= tmp;
}//---------------------
```

这样的类层次更准确地描述了现实世界。现实中确实有账户这一概念,也确实有储蓄类账户和结算类账户之分。

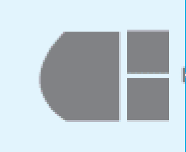
而且,Savings 类和 Checking 类互不干涉。Checking 类不再受 Savings 类的影响,反之亦然。如果银行对账户类有所改变,则可以修改 Account 类,如果只改变结算类政策,则只需修改 Checking 类即可,而储蓄类保持不变。

从相似的类中,将共有特征提取出来的过程称为分解(factoring)。分解使类的层次合理化和减少冗余。只有当继承关系与实际相符合时,分解才是合理的。

一个程序员努力工作,写出比较巧妙的代码,以达到减少一些程序行的目的,是不值得的。这种巧妙常常会弄巧成拙。但是通过继承而分解出多余部分,可以合理地减少编程的工作量。

17.7 抽象类与纯虚函数

分解带来了很多好处,但同时也产生了一个问题。在定义 Account 类的成员函数时,大多数成员函数不存在问题,因为两种账户都以相同的方式完成。



但 Withdrawal()就不同了。由于从一个储蓄账户中取款的操作与从一个结算账户中取款不同，即 Savings::Withdrawal()和 Checking::Withdrawal()的实现不同，那么 Account::Withdrawal()的定义该如何呢？

要新建一个账户，总是先要确定是储蓄账户还是结算账户，否则银行不知如何办理和不予办理。所有的账户要么是储蓄账户要么是结算账户，而一个 Account 账户仅仅是对这两个具体账户的共性进行分解而得到的一个抽象。Account 类是不完整的，因为它缺少具体账户的操作(专指 Withdrawal())。

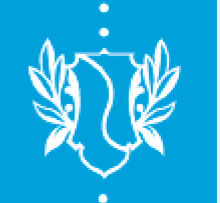
对于我们人类，从中可以分出中国人、美国人、德国人、埃及人等，中国人中还可以分出汉族和各种少数民族。一个人，他(她)必定属于世界上的某个国家和某个民族。脱离国家和民族的"纯粹"的人是没有的。人类是我们创造的一个高度抽象的概念，并不存在人类本身的实例。

我们不希望程序员建立 Account 类或人类对象。因为我们并不知道用它做什么。为了解决这个问题，C++允许程序员声明一个不能有实例对象的类，这样的类唯一的用途是被继承。这种类称为抽象类(abstract class)。

一个抽象类至少具有一个纯虚函数。所谓纯虚函数(pure virtual function)是指被标明为不具体实现的虚成员函数。例如，我们并不知道怎样实现 Account::Withdrawal()。然而又不得不给它一个定义，像上节中 account. cpp 中描述的"不做任何事"那样，否则，C++认为是缺定义成员函数的链接错误。

声明一个函数是纯虚函数的语法，即让 C++知道该函数无定义，在 Account 类中示例如下：

```
//----------------------
// account.h
//----------------------
#ifndef ACCOUNT
#define ACCOUNT
//----------------------
class Account{
public:
   Account(unsigned acntNo, float balan = 0.0);
   unsigned AccountNo(){ return acntNumber; }
   float AcntBalan(){ return balance; }
   static Account* First();                            //取链首指针
   Account* Next();                                    //取结点指针
   static int NoAccounts();
   void Display()const;
   void Deposit(float amount){ balance += amount; }
   virtual void Withdrawal(float amount) = 0;          //纯虚函数
protected:
   static Account* pFirst;
   static Account* pTail;
   Account* pNext;
   static int count;
   unsigned acntNumber;
   float balance;
}; //-----------------------------------
#endif // ACCOUNT
```



在 Withdrawal()的声明之后的"=0"表明程序员将不定义该函数。该声明是为派生类而保留的位置。Account 的派生类被期待用一个具体的函数来重载该函数。

一个抽象类不能有实例对象,即不能由该抽象类来制造一个对象。所以,下面的声明是非法的:

```
void func()
{
   Account a(3145, 300.0);      //企图建立对象: 账号为 3145,金额为 300 元
   a.Withdrawal(100);           //企图取 100 元
}
```

假如定义 Account 类对象允许,其结果是不完备的,缺少取款能力,因为并不存在 Account::Withdrawal()的具体行为。

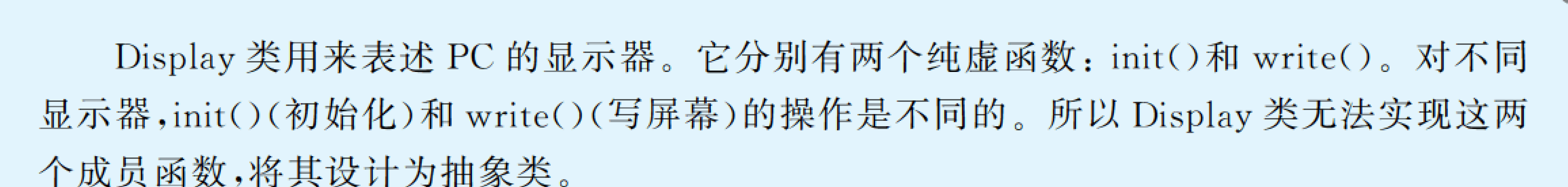
抽象类是作为基类为其他类服务的。一个 Account 类包含一个银行账户的所有特征。可以通过继承 Account 来创建其他类型的银行账户类,但是 Account 自身不能有实例对象。

17.8 抽象类派生具体类

Savings 类不是抽象类,因为它用一个完全实在的定义对纯虚函数 Withdrawal()进行了重载。Savings 类的一个对象知道在 Withdrawal()被调用时怎样执行。同样,Checking 类也如此,该类不是抽象的,因为成员函数 Withdrawal()重载了基类中的纯虚函数。

所有纯虚函数被重载之前,抽象类的子类也一直保持抽象状态。例如:

```
class Display                                //显示器类(抽象类)
{
  public:
    virtual void init() = 0;                 //纯虚函数
    virtual void write(char * pString) = 0;  //纯虚函数,输出字符串
};
class Monochrome : public Display            //单色显示器类
{
  public:
    virtual void init(){}                    //虚函数定义
    virtual void write(char * pString){}     //虚函数定义
};
class ColorAdapter : public Display          //彩色显示器类(抽象类)
{
  public:
    virtual void write(char * pString){}     //虚函数定义
};
class SVGA : public ColorAdapter             //VGA 系列彩色显示器类
{
  public:
    virtual void init(){}                    //虚函数定义
};
void func()
{
  Monochrome rac;                            //ok
  SVGA vga;                                  //ok
}
```



Display 类用来表述 PC 的显示器。它分别有两个纯虚函数：init()和 write()。对不同显示器，init()(初始化)和 write()(写屏幕)的操作是不同的。所以 Display 类无法实现这两个成员函数，将其设计为抽象类。

Display 类的子类 Monochrome 不是抽象类。这是一个特定的显示器类型，程序员知道该如何编程。因此，定义了 init()和 write()这两个虚函数。

SVGA 类也不是抽象类，程序员也知道怎样对该显示器编程。然而，在介于 Display 类和 SVGA 类之间，有一个 ColorAdapter 类，SVGA 是所有彩色显示器的一个特例。该类能够确定如何写屏(实现了成员函数 write())，这里假定了所有的彩色显示器都以相同的方法进行写屏。但对不同的彩色显示器，还是不能决定如何初始化。因此其 init()成员函数从 Display 类中继承过来，还是纯虚函数。所以，它也仍然是抽象类。

SVGA 类中实现了 init()成员函数，使之不再具有纯虚函数，所以，SVGA 是可以创建具体对象的实在类。

17.9 多态的目的

1. 抽象类指针参数

不能创建一个抽象类的对象，但是可以声明一个抽象类的指针或引用。

例如，下面代码中的类，取自 17.7 节中的 Account 抽象类及其派生的类族。设计一个专门处理账户的函数：

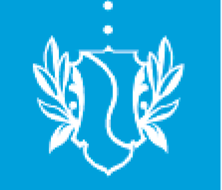
```
void func(Account * pA)   //专门处理账户中的取款参数为抽象素的指针
{
   pA->Withdrawal(100.0);
}

void otherFunc()
{
   Savings s;
   Checking c;
   func(&s);              //合法：一个 Savings 是一个 Account
   func(&c);              //合法：一个 Checking 也是一个 Account
}
```

在这里，函数 func()的参数 pA 是一个 Account 指针。从 otherFunc()函数调用 func()时，传递的实参都是具有实际地址的子类对象。它们要么是 Savings 类对象，要么是 Checking 类对象，但绝不可能是 Account 类对象，因为在 otherFunc()函数中，绝不可能创建 Account 对象。

在分解银行存款的例子中，**如果在 Account 类里去掉 Withdrawal()纯虚函数，使得在其子类中分别添加 Withdrawal()成员函数的定义，程序仍能通过编译和链接**，在某些情况下也能运行。

但若考虑上例混搭处理各子类对象的多态的情形，则下例的代码将不能通过编译：




```
class Account
{
   //除了不声明 Withdrawal()外,其他都一样
};

class Savings :public Account
{
   public:
   virtual void Withdrawal(float amount);
};

//...

void func(Account * pA)
{
   pA->Withdrawal(100.0);   //error:Withdrawal()不是 Account 的成员
}

int main()
{
   Savings s;
   func(&s);
}
```

C++是强类型语言。当访问一个成员函数时,C++坚持要证明该成员函数在类中存在,否则拒绝接受。在上例的 func()函数中,pA 指针是 Account 类的,在编译时,就将验证所指向的函数 Withdrawal()是否为其成员。如果 Withdrawal()是 Account 的成员函数,即使是纯虚函数,也能顺利通过编译。这正是纯虚函数为什么非要不可的原因所在。

纯虚函数是在基类中为子类保留的一个位置,以便子类用自己的实在函数定义来覆盖之。如果在基类中没有保留位置,则无法覆盖。

2. 批量处理类族对象

因为继承,所以有了类族对象,有了批量处理类族对象的需求。而多态便是专为处理类族对象而设。我们准备了数据文件 acc.txt,如下所示:

```
S 12345 12345.00
S 10000 10000.00
S 09988 9988.00
C 20032 20032.00
C 25678 25678.00
S 26100 26100.00
```

数据中的每一行代表一个账户信息,第一项中的 S 表示 Savings 账户,C 表示 Checking 账户,第二项表示账号,第三项表示余额。

我们改善一下 17.6 节中的 Account 类,以及它的子类 Savings 和 Checking 类。将 Account 类设计为抽象类,完善其链表操作以便更容易进行应用编程。然后,让主程序读取文件数据中的一叠数据,不管是来自 Savings 还是 Checking 的账户,依赖 Account 抽象类,建立诸银行账户的一个账户链表。我们的目的是要多态处理该账户链表,而不论账户品种如何多样复杂。

```
//----------------------
// account.h
//----------------------
#ifndef ACCOUNT
#define ACCOUNT
//----------------------
class Account{
public:                                          //链表处理部分
  Account(unsigned acntNo, float balan = 0.0);
  static Account* First(){ return pFirst; }      //取链首指针
  Account* Next(){ return pNext; }               //取结点指针
  static void RemoveLink();
public:                                          //账户处理部分
  unsigned AccountNo(){ return acntNumber; }
  float AcntBalan(){ return balance; }
  void Deposit(float amount){ balance += amount; }
  virtual void Display()const;                   //虚函数
  virtual void Withdrawal(float amount) = 0;     //纯虚函数
protected:                                       //链表成员
  static Account* pFirst;
  static Account* pTail;
  Account* pNext;
protected:                                       //账户数据
  unsigned acntNumber;
  float balance;
}; //-----------------------------------
#endif // ACCOUNT

//----------------------
// account.cpp
//----------------------
#include"account.h"
#include<iostream>
using namespace std;
//----------------------
Account* Account::pFirst = 0;                    //链表为空
Account* Account::pTail = 0;
//----------------------
Account::Account(unsigned acntNo, float balan)
  :acntNumber(acntNo), balance(balan){
  if(pFirst == 0)
    pFirst = this;
  else
    pTail->pNext = this;
  pTail = this;
  pNext = 0;
}//---------------------
void Account::Display()const{
  cout <<"Account number:"<< acntNumber <<" = "<< balance <<"\n";
}//---------------------
void Account::RemoveLink(){
  for(Account *q, *p = Account::First(); p; p = q){
    q = p->pNext;
    delete p;
```




```
    }
}//---------------------

//----------------------
// savings.h
//----------------------
#ifndef SAVINGS
#define SAVINGS
//----------------------
#include"account.h"
//----------------------
class Savings : public Account{
public:
    Savings(unsigned acntNo, float balan = 0.0);
    virtual void Withdrawal(float amount);      //虚函数
    virtual void Display()const;
protected:
    static float minbalance;
}; //-------------------------------------
#endif // SAVINGS

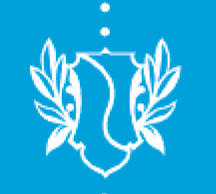
//----------------------
// savings.cpp
//----------------------
#include"savings.h"
#include"account.h"
#include<iostream>
using namespace std;
//----------------------
float Savings::minbalance = 500;                //设置透支范围
//----------------------
Savings::Savings(unsigned acntNo, float balan)
    : Account(acntNo, balan){}
//----------------------
void Savings::Display()const{
    cout <<"Savings ";
    Account::Display();
}//---------------------
void Savings::Withdrawal(float amount){
    if(balance + minbalance < amount)
        cout <<"Insufficient funds: balance "<< balance <<",withdrawal"<< amount <<"\n";
    else
        balance -= amount;
}//---------------------

//----------------------
// checking.h
//----------------------
#ifndef CHECKING
#define CHECKING
//----------------------
#include"account.h"
//----------------------
enum REMIT{remitByPost, remitByCable, other}; //信汇, 电汇, 无
```

```
//----------------------
class Checking : public Account{
protected:
  REMIT remittance;
public:
  Checking(unsigned acntNo, float balan = 0.0);
  void Withdrawal(float amount);
  void Display()const;
  void setRemit(REMIT re){ remittance = re; }
}; //--------------------
#endif // CHECKING

//----------------------
// checking.cpp
//----------------------
#include"checking.h"
#include"account.h"
#include<iostream>
using namespace std;
//----------------------
Checking::Checking(unsigned acntNo, float balan)       //基类初始化完成链表操作
  : Account(acntNo, balan), remittance(other){}
//----------------------
void Checking::Withdrawal(float amount){
  float tmp = amount;
  if(remittance == remitByPost)                        //信汇加收 30 元手续费
    tmp = amount + 30;
  if(remittance == remitByCable)                       //电汇加收 60 元手续费
    tmp = amount + 60;
  if(balance < tmp)
    cout <<"Insufficisent funds:balance "<< balance <<", withdrawal"<< tmp <<"\n";
  else
    balance -= tmp;
}//---------------------
void Checking::Display()const{
  cout <<"Checking ";
  Account::Display();
}//---------------------

//----------------------
// ch17_5.cpp
//----------------------
#include"account.h"
#include"savings.h"
#include"checking.h"
#include<iostream>
#include<fstream>
using namespace std;
//----------------------
void getLink();
void doBusiness(Account * p);
//----------------------
int main(){
  getLink();                                           //获取链表
```




```
    for(Account * p = Account::First(); p; p = p->Next())        //获取链首指针,遍历链表
      doBusiness(p);                                              //释放链表
    Account::RemoveLink();
}//--------------------
void getLink(){
    ifstream cin("acc.txt");
    char c;
    unsigned int ac;
    for(float blan; cin >> c >> ac >> blan; )                     //每个结点插入链首
      if(c == 'S') new Savings(ac,blan);
      else new Checking(ac,blan);
}//--------------------
void doBusiness(Account * p){
    p->Withdrawal(2);                                             //收取月费
    p->Display();
}//--------------------
```

运行结果为:

```
Savings Account number:12345 = 12343
Savings Account number:10000 = 9998
Savings Account number:9988 = 9986
Checking Account number:20032 = 20030
Checking Account number:25678 = 25676
Savings Account number:26100 = 26098
```

主函数是通过调用 getLink()函数来创建链表的。getLink()函数循环读入文件数据,判断不同账户,从堆中申请空间创建 Savings 或 Checking 对象,创建的同时即调用相应的构造函数,从而挂接到账户链表尾部。因为在堆中创建,由链表维护者 Account 抽象类的静态成员函数 RemoveLink()负责最后的链表释放善后工作,所以并没有单独设计子类的析构函数。

主函数通过反复调用 doBusiness()函数,对多态进行展示。显然,一个指针链表承载着一叠类族中的异类对象。但是它们的结点又同属一种 Account * 类型,使得循环(批量)处理异类对象成为可能。

而带有指针 Account * 参数的函数 doBusiness()接应着来自同一类族的异类对象的指针传递,通过调用 Withdrawal()和 Display()函数,展现了多态。

小结

C++虚函数技术是实现多态的重要手段。继承从基类开始,虚函数也是从基类开始往下传播。类家族成员的各种对象因而可展现出多态。

继承的合理设计能够保证类的层次性和可维护性,从而确立一个稳固的抽象的基类,于是带有纯虚函数的抽象类便应运而生。

纯虚函数是一个没有定义函数语句的虚函数,往往设在基类,纯虚函数的值一定是 0,可以理解为函数空指针,作为具体类的子类必须为每一个基类纯虚函数提供一个相应的函数定义,才可以创建自己的对象。



练习

17.1 根据 17.6 节中所定义的 Account 类、Savings 类和 Checking 类，编写一个应用程序，它读入一系列账号和存款，创建若干储蓄和结算账户，直到碰到一个标志结束的符号，并输出所有账号的存款数据。

17.2 根据 17.6 节中所定义的 Account 类、Savings 类和 Checking 类，编写一个应用程序，它取出一系列账号的存款，直到碰到一个标志结束的符号。要求程序用多态的方法实现，并输出取出的账号和金额数。

17.3 将习题 17.1 和 17.2 中的应用程序设计成函数，设计一个菜单，选择处理储蓄和结算账户；还需在子菜单中选择处理存款和取款业务，并分别调用上面设计的两个函数。

17.4 用多文件程序结构实现习题 17.3，画出其类层次图。

17.5 信用卡是储蓄类的一种，假设它可以在 5000 元范围内透支，它有一个用户密码，取款时，必须验证密码。从 Account 账户类体系中继承一个信用卡类，然后编制应用程序，实现取款和存款业务，并将其纳入习题 17.3 程序的菜单之中。

17.6 定期储蓄是储蓄的一种，假设定期分一年期、三年期和五年期，利率分别为 5%、8% 和 10d。用户在办理定期存款账户时，必须确定其定期时段，中途不再在同一账号上办理存款业务。取款是一次性完成，若提前取款，则全部金额的利息按活期利率 1% 计算。将其银行业务纳入习题 17.3。

17.7 专用存款是结算类账户的一种，银行为了监督专用存款的使用，特地将该存款另设账户进行管理。专用存款的存取业务与结算类账户相同。从银行账户体系中派生专用存款账户，并将其应用纳入习题 17.3 中。

第18章　运算符重载

重载运算符是 C++的一个特性，它使得程序员可把 C++运算符的定义扩展到操作数是对象的情况。运算符重载的目的是：使 C++代码更直观，更易读，由简单的运算符构成的表达式常常比函数调用更简洁、易懂。学习本章后，应该理解怎样重定义与类有关的运算符，学会怎样把一个类对象转换为另一个类对象，能把握重载运算符的时机。

18.1　运算符重载的需要性

运算符即操作符，见表 3-1 所列。

C++认为用户定义的数据类型就像基本数据类型 int 和 char 一样有效。运算符（如＋、－、* 、/）是为基本数据类型定义的，为什么不允许它也适用于用户定义类型呢？例如：

```
class A
{
   int a;
   public:
     A(int x)
     {
       a = x;
     }

   //
};

A a ( 5 ), b ( 10 ), c;
c = a  +  b;      //类对象也应能运算
```

运算符重载可以改进可读性，但不是非有不可。

下列例子计算应付人民币，分别用了成员函数和运算符成员函数两种方法：

```
//----------------------
//   ch18_1.cpp
//----------------------
```

```
#include<iostream>
using namespace std;
//--------------------
class RMB{                          //人民币类
public:
  RMB(double d){ yuan=d; jf=(d-yuan)/100; }
  RMB interest(double rate);        //计算利息
  RMB add(RMB d);                   //人民币加
  void display(){ cout<<(yuan + jf / 100.0)<<endl; }
  RMB operator+(RMB d){ return RMB(yuan+d.yuan+(jf+d.jf)/100); } //重载人民币加
  RMB operator*(double rate){ return RMB((yuan+jf/100)*rate); }
private:
  unsigned int yuan;                //元
  unsigned int jf;                  //角分
}; //--------------------
RMB RMB::interest(double rate){
  return RMB((yuan + jf / 100.0) * rate);
}//--------------------
RMB RMB::add(RMB d){
  return RMB(yuan+d.yuan + jf / 100.0 + d.jf / 100.0);
}//--------------------
//以下是计算应付人民币的两个版本
RMB expense1(RMB principle, double rate){
  RMB interest=principle.interest(rate);
  return principle.add(interest);
}//--------------------
RMB expense2(RMB principle, double rate){
  RMB interest=principle * rate;    //本金乘利息
  return principle+interest;        //连本带利
}//--------------------
int main(){
  RMB x=10000.0;
  double yrate=0.035;
  expense1(x,yrate).display();
  expense2(x,yrate).display();
}//--------------------
```

运行结果为：

```
10350
10350
```

expense()的两个版本都可以计算应付人民币，运行结果相同。expense2()可读性更好一点，它符合我们计算用+、*运算符的习惯。

如果不定义运算符重载，则 expense2() 中 principle * rate 和 principle + interest 是非法的。因为参加运算的操作数是类对象而不是浮点值。

18.2 如何重载运算符

1. 优先级与结合性不变

运算符是在 C++ 系统内部定义的，具有特定语法规则，如参数说明、运算顺序、优先级



等。重载运算符时,要注意该重载运算符的运算顺序和优先级不变,如下例:

```
class A
{
   public:
     A(int n)
     {
       //...
     }
     friend A operator + (A&, A&)
     {
       //...
     }
     friend A operator * (A&, A&)
     {
       //...
     }

     //...
};

A a=5, b=6, c=7, d=8, e;
e=a+b*c+d;      // 即 (a+(b*c))+d
```

有了运算符,编程就显得方便。例如,对于直角三角形斜边长度公式 $c=\sqrt{a^2+b^2}$,用函数化的格式表示:

```
c=sqrt ( add ( mult ( a, a ), mult ( b, b ) ) );
```

用运算符的格式表示更简洁易读:

```
c=sqrt ( a * a + b * b );
```

2. 操作数个数不变

运算符的操作数是规定好了的,例如,乘法和加法是双目运算符,++是单目运算符,等等。如果改变运算符的操作数个数,将带来编译器错误。例如:

```
class A{
public:
   A(int a){}
   A operator * (A& x, A& y, A& z)
   {   //error 试图带 3 个参数
     //...
   }
};
```

3. 操作数类型规定

运算符是函数,除了运算顺序和优先级不能更改外,参数和返回类型是可以重新说明的,即可以重载。重载的形式是:

```
返回类型 operator 运算符号(参数说明);
```

例如：A 类对象加法：

```
class A{ };

int operator + (A&, A&);      //两个 A 类对象参加运算,返回 int 型值
```

C++规定，运算符中参数说明都是内部类型时，不能重载。例如不允许声明：

```
int * operator + (int, int * );
```

即不允许进行下述运算：

```
int a = 5;
int * pa = &a;
pa = a * pa;       //error
```

C++基本数据类型之间的关系是确定的，如果允许定义其上的新操作，那么，基本数据类型的内在关系将混乱。

4. 不能重载的运算符

C++还规定了点操作符(.)、域操作符(::)、成员间访操作符(. *)、成员指针操作符(->*)、条件操作符(?:)，这五个运算符不能重载，也不能创造新运算符。例如，不允许声明：

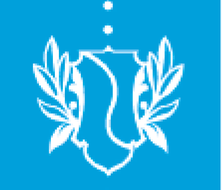
```
int operator @(int, int);
```

或者：

```
int operator ::(int, int);
```

例如，下面的程序将运算符+和++声明为人民币类的友元：

```
//---------------------
//  ch18_2.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class RMB{
public:
  RMB(unsigned int d, unsigned int c);
  friend RMB operator + (RMB&, RMB&);
  friend RMB& operator++(RMB&);
  void display(){ cout <<(yuan + jf / 100.0)<< endl; }
protected:
  unsigned int yuan;
  unsigned int jf;
}; //--------------------
RMB::RMB(unsigned int d, unsigned int c){
  yuan = d + c/100;
  c = c % 100; //构造时,确保角分值小于 100
}//--------------------
RMB operator + (RMB& s1, RMB& s2){
  unsigned int jf = s1.jf + s2.jf;
```




```
    unsigned int yuan = s1.yuan + s2.yuan + jf/100;
    jf = jf % 100;
    return RMB( yuan, jf );
  }//--------------------
  RMB& operator++(RMB& s){
    if(++s.jf == 100){
      s.jf = 0;
      s.yuan++;
    }
    return s;
  }//--------------------
  int main(){
    RMB d1(1, 60);
    RMB d2(2, 50);
    RMB d3(0, 0);
    d3 = d1 + d2;
    ++d3;
    d3.display();
  }//--------------------
```

运行结果为：

```
4.11
```

operator +() 和 operator ++() 定义为友元是为了能访问人民币类的保护成员。

operator +()是一个双目运算符，它有两个参数 s1 和 s2，并且相加的结果仍为人民币类，返回人民币类对象。

→ 不是必须要让 operator +()执行加法，可以让它做任何事。但是不让它做加法，而做其他操作是一个很糟的想法。如果重载＋运算符，向一个文件写 10 次“I like C++”，语法上可以，但与语义相差悬殊，不利于可读性，背离了允许运算符重载的初衷。当别人读这个程序时，发现 s1 ＋ s2 的操作，想象是某种加法操作，怎么也想不到会是这样的写操作。所以在使重载运算符脱离原义之前，必须保证有充分的理由。

→ 为什么 operator +()中的参数用引用传递而不用指针传递？

因为指针传递存在程序上的可读性问题。如果操作符重载声明为：

```
RMB operator + (RMB * a, RMB * b);
```

则调用时

```
RMB s1(5.1);
RMB s2(6.7);
RMB c = &s1 + &s2;      //是 s1 的地址与 s2 的地址相加吗?
```

operator++()是单目运算符，它含有一个参数。operator ++()对人民币类对象的角分做加 1 运算，如果它超过 100，则对该对象的元做加 1 运算并使角分为 0(减 100)。

如果只给出一个 operator++()定义，那么它一般可用于前缀、后缀两种形式。即 d3++ 与++d3 不作区别。

18.3 值返回与引用返回

上节中，为什么 operator +()由值返回，而 operator ++()由引用返回呢？

重载定义+和++操作的意义是人为的，所以返回类型并非一定如此规定。但如上节定义的+和++操作的意义，应该规定＋由值返回，++由引用返回。

对于 operator +()，两个对象相加，不改变其中任一个对象。而且它必须生成一个结果对象来存放加法的结果，并将该结果对象以值的方式返回给调用者。

如果＋以引用返回如下例：

```
RMB& operator + (RMB& s1, RMB& s2)
{
   unsigned int jf = s1.jf  +  s2.jf;
   unsigned int yuan = s1.yuan  +  s2.yuan;
   RMB result(yuan, jf);
   return result;
}
```

则尽管编译正确，能够运行，但会产生奇怪的结果。例中的 result 对象由+运算符函数的栈空间分配内存，受限于块作用域，引用返回导致了调用者使用这块会被随时分配的空间(见9.6节)。

能否将结果对象从堆中分配来避免上例的问题呢？例如：

```
RMB& operator + (RMB& s1, RMB& s2)
{
   unsigned int jf = s1.jf + s2.jf;
   unsigned int yuan = s1.yuan + s2.yuan;
   return  * new RMB(yuan, jf);
}
```

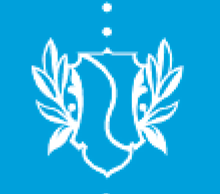
虽然它无编译问题，可以运行，但是该堆空间无法回收，因为没有指向该堆空间的指针，会导致内存泄漏，程序不断做加法时，堆空间也在不断流失。

如果坚持结果对象从堆中分配，而返回一个指针，那样在应用程序中就要付出代价：

```
void fn(RMB& a, RMB& b)
{
   RMB *  pc = a  +  b;      // c = a + b; 必须由此三条语句代替
   RMB c =  * pc;
   delete pc;
}
```

通过值返回，将有一个临时对象在调用者的栈空间产生，它复制被调函数的 result 对象，以便参加调用者中的表达式运算，对于“c＝a＋b;”，则 a＋b 的临时对象赋给 c，然后临时对象的作用域也结束了。

与 operator +()不一样，operator ++()确实修改了它的参数，而且其返回值要求是左值，这个条件决定了它不能以值返回。如果以值返回：




```
RMB operator ++ (RMB& s)
{
   s.jf ++ ;
   if(s.jf >= 100)
   {
     s.jf -= 100;
     s.yuan ++ ;
   }

   return s;
}

//...
RMB a(2, 50);
c = a ++ ;                  //ok
c = ++ a;                   //ok,a 为 2.52
c = ++ ( ++ a );            //error,a 为 2.53,理应 2.54
```

因为++a返回一个对象值,这个对象值并非a本身,是临时对象的值,它从形参s中拷贝而来,随后又进行了括号外的++操作,再次产生临时对象,将值赋给c。所以a本身只进行了一次++操作。

18.4 运算符作成员函数

一个运算符除了可以作为一个非成员函数实现外,还可以作为一个成员函数实现。例如,下面的程序将ch18_2.cpp中的+和++运算符改成作为成员予以实现:

```
//---------------------
//    ch18_3.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class RMB{
public:
RMB(unsigned int d, unsigned int c);
  RMBoperator + (RMB&);
  RMB&operator++ ();
  void display(){ cout <<(yuan + jf / 100.0)<< endl; }
  protected:
  unsigned int yuan;
  unsigned int jf;
}; //-------------------
RMB::RMB(unsigned int d, unsigned int c){
  yuan = d + c/100;
  jf = c % 100;
}//--------------------
RMB RMB::operator + (RMB& s){
  unsigned int c = jf + s.jf;
  unsigned int d = yuan + s.yuan + c/100;
c = c % 100;
return RMB(d,c);
```

```
}//--------------------
RMB& RMB::operator++(){
  if(++jf == 100)
  jf = 0;
yuan++;
  }
  return *this;
}//--------------------
int main(){
  RMBd1(1, 60);
  RMBd2(2, 50);
  RMBd3(0, 0);
  d3 = d1 + d2;
  ++d3;
  d3.display();
}//--------------------
```

运行结果为：

```
4.11
```

从中看出，作为成员的运算符比之作为非成员的运算符，在声明和定义时，形式上少一个参数。这是由于C++对所有的成员函数隐藏了第一个参数this(见11.4节)。

下面列出非成员和成员形式的运算符来进行比较：

```
RMB operator +(RMB& s1, RMB& s2)      //非成员形式
{
   unsigned int jf = s1.jf + s2.jf ;
   unsigned int yuan = s1.yuan + s2.yuan;
   RMB result(yuan, jf);
   return result;
}

RMB RMB::operator +(RMB& s)           //成员形式
{
   unsigned int c = jf + s.jf ;
   unsigned int d = yuan + s.yuan;
   RMB result(c, d);
   return result;
}
```

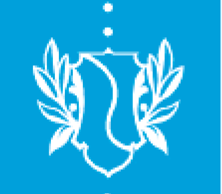
可见函数体中内容几乎相同，只是非成员形式加s1和s2，成员形式s加当前对象，当前对象的成员隐含着由this指向。即yuan意味着this->yuan。

一个运算符成员形式，将比非成员形式少一个参数，左边参数是隐含的。

作为人民币类的一种常规操作，我们应该允许其中有一个操作数是double型的情况：

```
c = c + 2.5; c = 2.7 + c;
```

但是由于参数类型不同，上例的运算符不论是成员形式还是非成员形式，都不能被这两个调用所匹配，还必须重载下列两个成员运算符：




```
RMB operator + (RMB& s, double d)
{
   unsigned int y = s.yuan + d;
   unsigned int j = s.jf + ( d - s.yuan ) * 100 + 0.5;
   RMB result(y, j);
}

inline RMB operator + (double d, RMB& s)
{
   return s + d;
}
```

这里第二个重载运算符调用了第一个重载运算符,两者之间只是参数顺序相反,定义后者为内联函数是一个技巧,省去了必要的开销。

从中得出,为了适应其中一个操作数是 double 的情况,不得不额外引入两个重载运算符。如果有构造函数:

```
RMB(double value)
{
   yuan = value;
   jf = ( value - yuan ) * 100.0 + 0.5;
}
```

就能够将 double 通过构造,变换成 RMB 类,于是:

```
class RMB
{
   public:
     RMB(unsigned int d, unsigned int c);
     RMB(double value);
  friend RMB operator + (RMB& s1, RMB& s2);

   //其余同前
};

int main ( )
{
   RMB s(5.8);
   s = RMB(1.5) + s;      //显式转换(创建一个无名对象)
   s = 1.5 + s;           //隐式转换
   s = s + 1.5;           //隐式转换
   s = s + 1;             //将 int 变换成 double,然后像上面那样变换
}
```

现在不必定义 operator +(double, RMB&) 和 operator +(RMB&, double)了,因为可将 double 转换成 RMB 类,然后匹配 operator +(RMB&, RMB&)。

该变换可以是显式的,如 s=RMB(1.5) + s 那样,也可以是隐含的。此时,由于其中的一个操作数是 RMB 对象,而且参数个数相同,所以它首先假定 operator +(RMB&, RMB&)可以匹配,然后寻找能够使用的转换。发现构造函数 RMB(double)可作为转换的依据。在完成转换后,真正匹配 operator +(RMB&, RMB&)运算符。所以程序员可以通过定义转换函数,来减少定义的运算符个数。



但是如果是下面的情况：

```
s=1.5+6.4;
```

那么由于左右操作数都是 double 型，所以匹配基本数据类型的加法，进行浮点运算。然后因为赋值表达式左面是 RMB 对象，所以该赋值运算将右面表达式的结果用构造函数 RMB(double)进行 RMB 转换，再赋值给 s。

C++规定：=、( )、[ ]、-> 这 4 种运算符必须为成员形式。

→ 我们在实现加法运算符时，用的是非成员形式。如果将运算符改成成员形式，那么，对于 s=1.5 + s；的形式，仍然必须要有重载运算符 RMB operator +(double, RMB&)来支持，因为在表达式 1.5+s 中，左面的 1.5 不能匹配 RMB::operator +(RMB&)中的隐含类对象，由于没有双目运算符的非成员形式，所以也无法利用类的转换来创造匹配的条件。这就是为什么有些运算符重载(如复数类的+、-运算)用非成员形式的原因。

18.5 重载增量运算符

在 ch18_2.cpp 中，描述的重载增量运算符是不区分前增量与后增量的。那么编译器是如何区分前增量和后增量的呢？

1. 前增量与后增量的区别

使用前增量时，对对象(操作数)进行增量修改，然后再返回该对象。所以前增量运算符操作时，参数与返回的是同一个对象。这与基本数据类型的前增量操作类似，返回的也是左值。

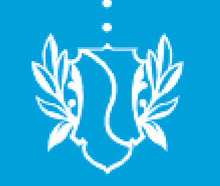
使用后增量时，必须在增量之前返回原有的对象值。为此，需要创建一个临时对象，存放原有的对象，以便对操作数(对象)进行增量修改时，保存最初的值。后增量操作返回的是原有对象值，不是原有对象，原有对象已经被增量修改，所以，返回的应该是存放原有对象值的临时对象。

2. 成员形式的重载

C++约定，在增量运算符定义中，放上一个整数形参，就是后增量运算符。

例如，下面的程序分别定义了前增量与后增量成员运算符：

```
//---------------------
//    ch18_4.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
class Increase{
public:
  Increase(int x):value(x){}
  Increase&operator++();            //前增量
  Increaseoperator++(int);          //后增量
  void display(){ cout <<"the value is "<< value << endl; }
```




```
  private:
    int value;
};//--------------------
Increase& Increase::operator++(){
    value++;                    //先增量
    return *this;               //再返回原对象
}//--------------------
Increase Increase::operator++(int){
    Increase temp(*this);       //临时对象存放原有对象值
    value++;                    //原有对象增量修改
    return temp;                //返回原有对象值
}//--------------------
int main(){
    Increasen(20);
    n.display();
    (n++).display();            //显示临时对象值
    n.display();                //显示原有对象
    ++n;
    n.display();
    ++(++n);
    n.display();
    (n++)++;                    //第二次增量操作对临时对象进行
    n.display();
}//--------------------
```

运行结果为:

```
the value is 20
the value is 20
the value is 21
the value is 22
the value is 24
the value is 25
```

前后增量操作的意义,决定了其不同的返回方式。前增量运算符返回引用,后增量运算符返回值。

后增量运算符中的参数 int 只是为了区别前增量与后增量,除此之外没有任何作用。因为定义中无须使用该参数,所以形参名在声明与定义中均省略。

对于(n++)++中的第二个++是对返回的临时对象所做的,从最后一行输出可以看出对 n 的修改只发生一次。

3. 非成员形式重载

前增量和后增量的非成员运算符,也有类似的编译区分方法。例如,下面的程序将 ch18_4.cpp 中的前增量和后增量运算符修改为非成员形式:

```
//---------------------
//    ch18_5.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
```

```
class Increase{
public:
  Increase(int x):value(x){}
  friend Increase& operator++(Increase& );      //前增量
  friend Increase operator++(Increase&,int);    //后增量
  void display(){ cout <<"the value is "<< value << endl; }
private:
  int value;
}; //--------------------
Increase &operator++(Increase & a){
  a.value++;                                    //前增量
  return a;                                     //再返回原对象
}//---------------------
Increaseoperator++(Increase& a, int){
  Increase temp(a);                             //通过拷贝构造函数保存原有对象值
  a.value++;                                    //原有对象增量修改
  return temp;                                  //返回原有对象值
}//---------------------
int main(){
  Increasen(20);
  n.display();
  (n++).display();                              //显示临时对象值
  n.display();                                  //显示原有对象
  ++n;
  n.display();
  ++(++n);
  n.display();
  (n++)++;                                      //第二次增量操作对临时对象进行
  n.display();
}//---------------------
```

运行结果为：

```
the value is 20
the value is 20
the value is 21
the value is 22
the value is 24
the value is 25
```

可见，前增量和后增量运算符的定义以及成员形式与非成员形式稍有不同，但前增量和后增量运算符的使用完全相同。

18.6 转换运算符

转换运算符的声明形式为：

```
operator 类型名();
```

它没有返回类型，因为类型名就代表了它的返回类型，故返回类型显得多余。

转换运算符将对象转换成类型名规定的类型。转换时的形式就像强制转换一样。如果没有转换运算符定义，直接用强制转换是不行的，因为强制转换只能对基本数据类型进行操



作,对类类型的操作是没有定义的。

例如,下面的程序在类中定义了转换运算符,在主函数中将 double 数分别显式和隐式转换成 RMB 对象:

```
//----------------------
//    ch18_6.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
class RMB{
public:
  RMB(double value = 0.0);
  operator double(){ return yuan + jf / 100.0; }     //转换运算符
  void display(){ cout <<(yuan + jf / 100.0)<< endl; }
protected:
  unsigned int yuan;
  unsigned int jf;
}; //--------------------
RMB::RMB(double value){
  yuan = value;
  jf = ( value - yuan ) * 100 + 0.5;
}//---------------------
int main(){
  RMBd1(2.0), d2(1.5), d3;
  d3 = RMB((double)d1 + (double)d2);                 //显式转换
  d3 = d1 + d2;                                      //隐式转换
  d3.display();
}//---------------------
```

运行结果为:

```
3.5
```

对于 d3=d1 + d2,C++系统依次:

(1) 寻找成员函数的+运算符(此处未找到);

(2) 寻找非成员+运算符(此处未找到);

(3) 由于存在内部运算符 operator +(double, double);所以假定匹配其程序中的加法;

(4) 寻找能将实参(RMB 对象)转换成 double 型的转换运算符 operator double()(找到)。

于是,d1、d2 转换成 double 型,匹配内部的 double 加法,得到一个 double 的结果值,然后,对左面是 RMB 对象的赋值运算符,将右面的表达式转换成 RMB 临时对象,赋值给 RMB 对象 d3。

转换运算符的优点:

有了转换运算符,不必提供对象参数的重载运算符。可以从转换路径,到达 double 型,进行基本运算,得到 double 结果,再构造回来。

转换运算符的缺点:

无法定义其类对象运算符操作的真正含义,因为转换之后,只能进行其他类型的运算符操作(如 double 加法运算);

通过提供一个 double 转换,所有甚至无意义的 RMB 运算也将获得 double 转换而得以

可操作。

转换运算符与转换构造函数(简称转换函数)互逆。例如,RMB(double)转换构造函数将 double 转换为 RMB,而 RMB::operator double()将 RMB 转换成 double。

除此之外,还要防止同一类型提供多个转换路径(转换的二义性),它会导致编译出错。例如,下面的代码将使编译出错:

```
class A
{
   public:
    A(B & b);              //用 B 类对象构造 A 对象(B 类对象转换成 A 类对象)
   //...
};

class B
{
   public:
    operator A();          //将 B 类对象转换成 A 类对象
   //...
};

int main()
{
   B sb;
   A a = A(sb);            //A(sb)是构造还是转换?
}
```

遇到 A(sb)时,编译找到 A 的转换函数,准备将其转换成 A 对象,可是又从 B 类找到转换运算符,也可转换成 A 对象,由于多义性,编译报错。

18.7 赋值运算符

1. 为什么要赋值运算符

只要是用户定义的类或结构,都应能进行赋值运算,这也是继承了 C 语言:

```
struct S { int a, b; };

S a, b;
a = b;      //C 语言允许如此赋值
```

→ 数组名不能赋值,一个数组名代表一个数据类型的实体集合,实质上是一个常量指针,所以它不能赋值:

```
int a[5];
int b[] = {3, 4, 5, 6, 7};
a = b; //error
```

对任何类,像拷贝构造函数一样,C++也默认提供赋值运算符,但要区别拷贝构造函数和赋值运算符:

```
void fn(MyClass& mc)
{
   MyClass newMC = mc;      //这是拷贝构造函数
   newMC = mc;              //这是赋值运算符
}
```

当拷贝构造函数执行时,newMC 对象还不存在,拷贝构造函数起初始化的作用。当赋值运算符在 newMC 上执行时,它已经是一个 MyClass 对象。

在拷贝构造函数中,我们碰到浅拷贝和深拷贝的问题(见 14.6 节),赋值运算符也同样,什么时候浅拷贝不合适,就应提供成员赋值运算符。

2. 如何重载赋值运算符

重载赋值运算符与重载其他运算符类似。

例如,下面的程序提供了赋值运算符作为 Name 类的公共成员,以使主函数(普通函数)中两个对象之间允许互相赋值:

```
//----------------------
//    ch18_7.cpp
//----------------------
#include <string>
#include <iostream>
using namespace std;
//----------------------
class Name{
public:
  Name(){ pName = 0; }
  Name(char* pn){ copyName(pn); }
  Name(Name& s){ copyName(s.pName); }
  ~Name(){ deleteName(); }
  Name&operator = (Name& s){          //赋值运算符
  deleteName();
  copyName(s.pName);
  return *this;
  }
void display(){ cout << pName << endl; }
protected:
  void copyName(char* pN);
  void deleteName();
  char* pName;
}; //--------------------
void Name::copyName(char* pN){
  pName = new char[strlen(pN) + 1];
  if(pName)
  strcpy(pName, pN);
}//--------------------
void Name::deleteName(){
  if(pName){
  delete pName;
pName = 0;
  }
}//--------------------
```

```
int main(){
  Names("claudette");
  Namet("temporary");
  t.display();
  t = s;                //赋值
  t.display();
}//--------------------
```

运行结果为：

```
temporary
claudette
```

Name类在存储区中保留了一个人的名字，在构造函数中该存储区是从堆中分配来的，存在浅拷贝问题，必须自定义赋值运算符与拷贝构造函数。

赋值运算符以operator =()的名称出现，看起来像一个析构函数后面跟着拷贝构造函数。通常赋值运算符有两部分，第一部分与析构函数类似，在其中取消对象已经占用的资源。第二部分与拷贝构造函数类似，在其中分配新的资源。

对象t创建时，具有名字“temporary”，它在堆中存放。在t＝s赋值过程中，通过调用deleteName()，原先名字占用的空间还给堆，再另外调用copyName()从堆中分配新存储区去存储新名字“claudette”。

拷贝构造函数不需要调用deleteName()，因为刚创建时，还没有分配存放名字的堆空间。

赋值运算符operator =()的返回类型是Name&。这与赋值的语义相匹配。C++要求赋值表达式左边的表达式是左值，它能进行诸如下列的运算：

```
int a, b = 5;
(a = b) ++ ;       //结果 a 为 6
```

如果一个类定义了++运算符，则它也能执行类似上面的表达式，得到正确的a值。例如，在ch18_3.cpp中如果增加一个人民币类的赋值运算符，且不返回引用：

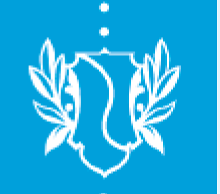
```
RMB operator = (RMB& s)
{
   yuan = s.yuan;
   jf = s.jf;
}
```

这时执行下列表达式：

```
RMB a(5.2), b(2.6);
(b = a) ++ ;      //结果 b 为 5.2,并不是期望的 5.3
```

因为b＝a返回的是对象值，而RMB类定义了++操作，所以，“(b＝a)++;”是合法的。但由于返回的不是引用，该值是b对象的一个复制，并不是b本身，所以++的操作数是b的复制对象而已。

→ *如果赋值运算符说明为保护或私有的，则可以将赋值操作限定在类的作用域范围，防止应用程序中使用赋值操作：*




```
class Name
{
  //
  protected:
    Name& operator = (Name& s)
    {
      //...
      return * this;
    }
};

//...
void fn(Name& n)
{
  Name newN;
  newN = n;        //error
}
```

因为 Name 类对象 newN 使得=匹配为 Name 类的赋值运算符，但是 protected 限定符使之不能在普通函数中被调用，从而防止了对象非法赋值操作。

小结

使用运算符重载可以使程序易于理解并易于对对象进行操作。几乎所有的 C++运算符都可以被重载，但应注意不要重载违反常规的运算符，不能改变运算符操作数的数量，也不能发明新运算符。

如果在类中没有说明本身的拷贝构造函数和赋值运算符，编译程序将会提供，但它们都只是对对象进行成员浅拷贝。在那些数据成员是指向堆空间指针的类中，必须避免使用浅拷贝，而要为类定义自己的赋值运算符，以给对象分配堆内存。

this 指针指向当前的对象，它是所有成员函数的不可见的参数，在重载运算符时，经常返回 this 指针的间接引用。

通过转换运算符可以在表达式中使用不同类型的对象。转换运算符不遵从函数应有返回值类型的规定，与构造函数和析构函数相同，它没有返回值。

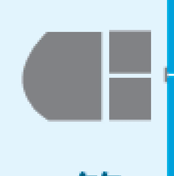
在前增量和后增量运算符定义中，使用 int 形参只是为了标志前后有别，没有其他作用。

拷贝构造函数用已存在的对象创建一个相同的新对象。而赋值运算符用已存在的对象赋予一个已存在的同类对象。

练习

18.1　定义复数类的加法与减法，使之能够执行下列运算：

```
Complex a(2, 5), b(7, 8), c(0, 0);
c = a + b;
c = 4.1 + a;
c = b + 5.6;
```



18.2 编写一个时间类，实现时间的加、减、读和输出。

18.3 根据 ch18_3.cpp，增加操作符，以允许人民币与 double 型数相乘。

```
friend money operator * (const money&, double);
    friend money operator * (double, const money&);
```

注意：两个 money 对象不允许相乘。

18.4 根据 ch18_3.cpp，增加操作符，以允许作相应赋值。

```
money& operator += (const money&);
money& operator += (double);
money& operator -= (const money&);
money& operator -= (double);
```

第19章 I/O流

C++的 I/O 流类,是最常用的 I/O 系统,到目前为止,我们一直在用这个类。学习了本章后,应该理解怎样使用 C++面向对象的 I/O 流,能够格式化输入和输出,理解 I/O 流类的层次结构,理解怎样输入和输出用户自定义类型的对象,能够建立用户自定义的流操作符,能够确定流操作的成败,能够把输出流系到输入流上。

19.1 printf 和 scanf 的缺陷

1. 非类型安全

函数原型使编译系统对它进行必要的类型检查,免除了许多错误。但对于 printf()和 scanf(),它却毫无帮助。printf()和 scanf()所期望的参数个数与类型取决于包含在第一个参数中的信息,而这一信息对编译器是没有用的。编译器无法检查对 printf()和 scanf()的调用的正确性。

例如,下面的函数企图输入和输出异于格式符的数据:

```
#include<stdio.h>

int j = 10;
float f = 2.3;

void fn()
{
   printf("%d",f);
   scanf("%d",&f);
   scanf("%d",j);
   printf("%d","abcde");
}
```

在 int 型占两个字节的情况下,语句 printf("%d",f);只输出 f 变量中前 2 个字节的内容,并按 int 型数据格式进行解释;

语句 scanf("%d",&f);只输入到 f 变量中前 2 字节中,按 int 型格式进行存放,而后面



两个字节内容却没有改变；

语句 scanf("%d",j)；将键入值存放到地址为 0x000A 的内存空间中；

语句 printf("%d","abcde")；输出"abcde"的地址值，而不是想要的字串。

上面这些语句，用错了数据类型，而编译都能通过。为此，程序员将花更多的代价在程序运行中出现的错误诊断上。特别对于 scanf() 中的错误，往往是致命的。

2. 不可扩充性

printf()和 scanf()知道如何输入输出已知的基本数据类型值，但是，C++程序中大量的类对象，其输入输出格式是未预先定义的，这就希望输入输出语句能够更加灵活与可扩充。printf()和 scanf()却无能为力，它们既不能识别，也不能学会如何识别用户定义的对象。

例如，下面的函数企图输入和输出一个类的对象：

```
class A{ /*...*/ };
A a;

//...

void fn()
{
   printf("%?",a);       //?表示不知以什么格式符来识别 A 的对象
   scanf("%?",&a);
}
```

19.2 I/O 标准流类

1. 标准流的设备名

iostream 是 I/O 流的标准头文件。其对应的标准设备见表 19-1。

表 19-1 标准 I/O 流设备

C++名字	设　备	C 中的名字	默认的含义
cin	键盘	stdin	标准输入
cout	屏幕	stdout	标准输出
cerr	屏幕	stderr	标准错误
clog	打印机	stdprn	打印机

这表明 cout 对象的默认输出是屏幕，cin 对象的默认输入是键盘。

2. 原理

cout 是 ostream 流类的对象，它在 iostream 头文件中作为全局对象定义：

```
ostream cout(stdout);     //标准设备名作为其构造时的参数
```

ostream 流类对应每个基本数据类型都有友元，它们在 iostream 中声明：




```
ostream& operator <<(ostream& dest,char * pSource);
ostream& operator <<(ostream& dest,int source);
ostream& operator <<(ostream& dest,char source);
//等等
```

分析语句:

```
cout <<"My name is Jone";
```

cout 是 ostream 对象,≪是操作符,右面是 char * 类型,故匹配上面的"ostream& operator≪(ostream& dest, char * pSource);"操作符。它将整个字串输出,并返回 ostream 流对象的引用。如果是:

```
cout <<"this is " << 7;
```

则根据≪的运算优先级,可以看作为:

```
(cout <<"this is ") << 7;
```

由于"cout ≪"this is ""返回 ostream 流对象的引用,与后面的≪7 匹配了另一个"ostream& operator≪(ostream& dest,int source);"操作符,结果构成了连续的输出。

同理,cin 是 istream 的全局对象,istream 流类也有若干个友元:

```
istream& operator >>(istream& dest,char * pSource);
istream& operator >>(istream& dest,int source);
istream& operator >>(istream& dest,char source);
//等等
```

除了标准输入输出设备,还有标准错误设备 cerr。

当程序测试并处理关键错误时,不希望程序的错误信息从屏幕显示重定向到其他地方,这时使用 cerr 流显示信息。

例如,下面程序在除法操作不能进行时显示一条错误信息:

```
//---------------------
//    ch19_1.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
void fn(int a, int b){
  if(b == 0)
  cerr <<"zero encountered. "
        <<"The message cannot be redirected\n";
  else
  cout << a/b << endl;
}//--------------------
int main(){
  fn(20,2);
  fn(20,0);
}//--------------------
```

运行结果为:

```
c > ch19_1 > abc.dat
zero encountered. The message cannot be redirected.
```



文件 abc. dat 的内容为：

```
10
```

主函数第一次调用 fn()函数时，没有碰到除 0 运算，得到文件的写内容 10，第二次调用 fn()函数时，碰到除 0 运算，于是在屏幕上输出错误信息。**写到 cerr 上的信息是不能被重定向的**，它只能输出到屏幕。

19.3 文件流类

ofstream、ifstream 和 fstream 是文件流类，在 fstream 头文件中定义。其中，fstream 是 ofstream 和 ifstream 多重继承的子类。文件流类不是标准设备，所以没有 cout 那样预先定义的全局对象。文件流类定义的操作应用于外部设备，最典型的设备是磁盘中的文件。要定义一个文件流类对象，须定义文件名和打开方式。

类 ofstream 用于执行文件输出，该类有几个构造函数，其中最常用的是：

```
ofstream::ofstream(char * pFileName,
                   int mode = ios::out,
                   int prot = filebuf::openprot);
```

第一个参数是指向要打开的文件名，第二和第三个参数说明文件如何被打开。mode 是文件打开方式，它的选择项见表 19-2。

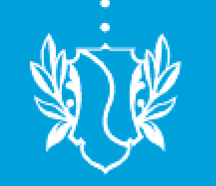
表 19-2　文件打开选择项

标　志	含　义
ios::ate	如果文件存在，输出内容加在末尾
ios::in	具有输入功能(ifstream 默认)
ios::out	具有输出功能(ofstream 默认)
ios::trunc	如文件存在，清除文件内容(默认)
ios::nocreate	如文件不存在，返回错误
ios::noreplace	如文件存在，返回错误
ios::binary	以二进制方式打开文件

prot 是文件保护方式，它的选择项见表 19-3。

表 19-3　文件保护方式选择项

标　志	含　义
filebuf::openprot	兼容共享方式
filebuf::sh_none	独占，不共享
filebuf::sh_read	允许读共享
filebuf::sh_write	允许写共享



例如,下面的程序在文件 myname 中写入一些信息:

```
//----------------------
//     ch19_2.cpp
//----------------------
#include <fstream>
using namespace std;
//----------------------
void fn(){
  ofstream myf("c:\\bctemp\\myname"); //默认 ios::out|ios::trunc
  myf <<"In each of the following questions, a related pair\n"
     <<"of words or phrases is followed by five lettered pairs\n"
   <<"of words or phrases.\n";
}//---------------------
int main(){
  fn();
}//---------------------
```

此处的文件名要说明其路径,斜杠要双写,因为编译器理解下的斜杠是转义字符。这与包含头文件时的路径不一样,因为包含头文件是由编译预处理器处理的。

文件打开时,匹配了构造函数 ofstream::ofstream(char *),只需一个文件名,其他为默认,打开方式默认为 ios::out|ios::trunc,即该文件用于接受程序的输出,如果该文件原先已存在,则其内容必须先清除,否则就新建。

例如,若要打开二进制文件,写方式,输出到文件尾,则:

```
ofstream bfile("binfile", ios::binary|ios::ate);
```

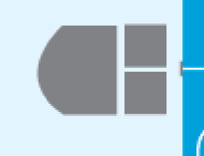
又例如,要检查文件打开否,则判断 fail()成员函数:

```
#include <fstream>
using namespace std;
void fn()
{
   ofstream myf("myname");
   if(myf.fail())      //fail() = =1 表示失败
   {
     cerr <<"error opening file myname\n";
     return;
   }
   myf <<"...";
}
```

例如,打开一个输入文件(要从文件中读数据):

```
#include <fstream>
using namespace std;
void fn()
{
   ifstream myinf("abc.dat", ios::nocreate);     //若文件丢失,就报错
   //...
}
```

可以通过检查 myinf.fail()来确定打开文件是否有错。



例如，打开同时用于输入和输出的文件：

```
fstream myinout("abc.dat", ios::in|ios::out);
```

用 ifstream 打开的文件，默认打开方式为 ios::in，用 fstream 打开的文件，打开方式不能默认。

19.4 C字串流类

ostrstream、istrstream 和 strstream 是 C 字串流类，在 strstream 头文件中定义。其中，strstream 是 ostrstream 和 istrstream 多重继承的子类。同样 C 字串流类不是标准设备，所以没有 cout 那样预先定义的全局对象。C 字串流类允许将 fstream 类定义的文件操作应用于存储区中的 C 字符串，即将字符串看作为设备，这很像 C 中的库函数 sprintf()和 sscanf()。要定义一个 C 字串流类对象，须定义字符数组和数组大小。

类 istrstream 用于执行 C 字串流输入，该类有几个构造函数，其中最常用的是：

```
istrstream::istrstream(const char * str);
istrstream::istrstream(const char * str, int size);
```

第一个参数指出字符串数组，第二个参数说明数组大小。当 size 为 0 时，表示把 istrstream 类对象连接到由 str 指向的以空字符结束的字符串。

例如，下面的代码定义一个 C 字串流类对象，并对其进行输入操作：

```
char str[100] = "I am a student.\n";
char a;
istrstream ai(str);        //将 str 看作输入设备
ai >> a;                   //从输入设备中输入一个字符
cout << a << endl;         //输出一个字符
```

输出结果为：

```
I
```

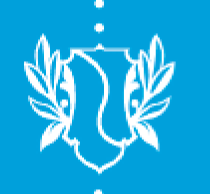
类 ostrstream 用于执行 C 字串流输出，该类也有几个构造函数，其中最常用的是：

```
ostrstream::ostrstream(char *, int size, int = ios::out);
```

第一个参数指出字符串数组，第二个参数说明数组大小，第三个参数指出打开方式。

例如，下面代码使用 C 字串流输入对字符串中的数据进行解读：

```
//----------------------
//    ch19_3.cpp
//----------------------
#include <iostream>
#include <strstream>
using namespace std;
//----------------------
char * parseString(char * pString){
  istrstream inp(pString);        //ios::in 方式
  int aNumber;
  float balance;
```




```
    inp >> aNumber >> balance;          //从 C 字串流中读入一个整数和浮点数

    char * pBuffer = new char[128];
    ostrstream outp(pBuffer,128);       //ios::out 方式,字串长度 128
    outp <<"a Number = "<< aNumber      //写入 pBuffer 中
        <<", balance = "<< balance;
  return pBuffer;
}//--------------------
int main(){
    char * str = "1234 100.35";
    char * pBuf = parseString(str);
    cout << pBuf << endl;
    delete[] pBuf;
}//--------------------
```

运行结果为:

```
a Number = 1234, balance = 100.35
```

在函数 parseString()中,以 pString 为输入设备,先定义一个输入 C 字串流对象 inp,从中输入一个整数和一个浮点数。

然后,开辟一个字符串空间(pBuffer 指向的 128 个字符)作为输出设备而定义输出 C 字串流对象 outp,将从输入设备中输入的该两个变量值输出。

19.5 控制符

C++有两种方法控制格式输出。

1. 用流对象的成员函数控制输出格式

例如,下面的程序改变有效位数:

```
//---------------------
//      ch19_4.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
void fn(float interest, float amount){
    cout <<"RMB amount = ";
    cout.precision(2);          //置有效位数为 2 位
    cout << amount;
    cout <<"\nthe interest = ";
    cout.precision(4);          //置有效位数为 4 位
    cout << interest << endl;
}//--------------------
int main(){
    float f1 = 29.41560067;
    float f2 = 12.567188;
    fn(f1,f2);
}//--------------------
```



运行结果为：

```
RMB amount = 13
the interest = 29.42
```

precision()为 cout 对象的成员函数，在要求输出一定精度的数据之前，先调用这个精度设置成员函数。

2. 用控制符控制输出格式

manipulators(控制符)是在头文件 iomanip 中定义的对象，与成员函数调用效果一样。控制符的优点是程序可以直接将它们插入流中，不必单独调用。

例如，用控制符设置有效位数，重写 ch19_4.cpp：

```
//---------------------
//    ch19_5.cpp
//---------------------
#include <iostream>
#include <iomanip>  //用到setprecision()
using namespace std;
//---------------------
void fn(float interest, float amount){
  cout <<"RMB amount = "
      << setprecision(2)<< amount;
  cout <<"\nthe interest = "
      << setprecision(4)<< interest << endl;
}//--------------------
int main(){
  float f1 = 29.41560067;
  float f2 = 12.567188;
  fn(f1,f2);
}//--------------------
```

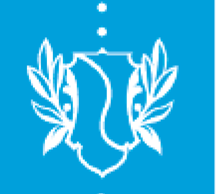
常用控制符和流对象成员函数如表 19-4 所示。

表 19-4 常用控制符与流对象成员函数

控　制　符	成员函数	描　　述
dec	flags(10)	置基数为 10
hex	flags(16)	置基数为 16
oct	flags(8)	置基数为 8
setfill(c)	flags(c)	设填充字符为 c
setprecision(n)	precision(n)	设显示小数精度为 n 位
setw(n)	width(n)	设域宽为 n 个字符

控制符和流成员函数相对应，它们用法不同，但作用相同。

其中 setw(n)或 width(n)很特别，它们在下一个域输出后，又回到原先的默认值。例如，输出下面的两个数：




```
//---------------------
//     ch19_6.cpp
//---------------------
#include <iostream>
#include <iomanip>   //用到setw()
using namespace std;
//---------------------
int main(){
  cout << setw(8)<< 10 << 20 << endl;
}//--------------------
```

运行结果为：

```
______1020
```

运行结果中的下横线表示空格。整数 20 并没有按宽度 8 输出。setw()的默认值为宽度 0，即 setw(0)，意思是按输出对象的表示宽度输出，所以 20 就紧挨 10 了。若要每个数值都有域宽度 8，则每个值都要设置：

```
cout << setw(8) << 10
     << setw(8) << 20 << endl;
```

从中得出，用控制符的方法更加直接。

又例如，下面的程序打印一个倒三角形：

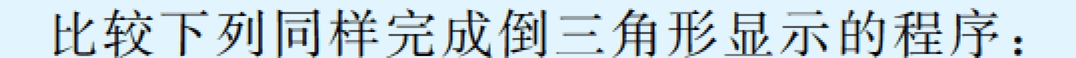
```
//---------------------
//     ch19_7.cpp
//---------------------
#include <iostream>
#include <iomanip>
using namespace std;
//---------------------
int main(){
for(int n = 1; n < 8; n++)
  cout << setfill(' ')<< setw(n)<<" "
    << setfill('m')<< setw(15 - 2 * n)<<"m"<< endl;
}//--------------------
```

运行结果为：

```
mmmmmmmmmmmmm
 mmmmmmmmmmm
  mmmmmmmmm
   mmmmmmm
    mmmmm
     mmm
      m
```

cout≪setfill(' ')≪setw(n)≪" "中所要显示的" "长度为 n，但它本身长度只有 1，所以其余的内容就由 setfill(' ')来填充了，效果就使得'm'前的空格逐行增加。同样，cout≪setfill('m') ≪setw(15－2 * n) ≪"m"中所要显示的"m"长度为 15－2 * n，但它本身长度只有 1，setfill('m')的作用就是将 15－2 * n 个空格用其'm'来填充，由于 15－2 * n 逐行递减，结果就显出一个用'm'构筑的倒三角形。

比较下列同样完成倒三角形显示的程序：



```
//----------------------
//     ch19_8.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  for(int n=1; n<=7; n++){
  for(int k=1; k<=n; k++)
    cout<<" ";
  for(int k=1; k<=15-2*n; k++)
    cout<<"m";
    cout<<endl;
  }
}//--------------------
```

但流成员函数也有其优点，它种类多，而且可以返回以前的设置，便于恢复设置。例如，下面的函数设置某有效位数输出，然后恢复成原来的有效位数设置：

```
//----------------------
//     ch19_9.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  float value=2.345678;
  int prePrecision=cout.precision(4);
  cout<<value;
  cout.precision(prePrecision);
  //...
}//--------------------
```

运行结果为：

```
2.346
```

假定程序中原来的有效位数设置不知道，“cout.precision(4)”可以返回原来设置的有效位数，保存该值在 prePrecision 变量中，使得最后用该值恢复原来的设置。

19.6 使用 I/O 成员函数

大多数简单的 C++程序使用 cin 进行输入操作。有时候需要对输入操作进行更加精细的控制，可以使用 I/O 流成员函数。

1. 用 getline()读取输入行

当程序使用 cin 输入时，cin 用空白符和行结束符将各个值分开。但根据所需输入的值，可能需要读取一整行文本包括空白符。为了读取整行文本，可以使用 getline 成员函数。



其成员函数原型为：

```
getline(char * line, int size, char = '\n');
```

第一个参数是字符数组，用于放置读取的文本；第二个参数是本次读取的最大字符个数；第三个参数是分隔字符，作为读取一行结束的标志，默认为回车符。

例如，下面的函数从键盘读取一行文本：

```
#include <iostream>
using namespace std;
void fn()
{
   char str[128];

   cout <<"Type in a line of text and press Enter" << endl;
   cin.getline(str, sizeof(str));
   cout <<"You typed: " << str << endl;
}
```

在默认状态，getline 成员函数读字符直到遇到回车，或者读到指定的字符数。如果在遇到某个字符(比如字母'X')时，需要结束输入操作，可以按下面方式使用：

```
cin.getline(line, sizeof(line), 'X');
```

例如，下面程序在遇到'X'字符处结束第一个输入操作，然后执行第二个输入：

```
//----------------------
//    ch19_10.cpp
//----------------------
#include <iostream>
using namespace std;
//----------------------
int main(){
  char str[128];
  cout <<"Type in a line of text and press Enter"<< endl;
  cin.getline(str, sizeof(str), 'X');
  cout <<"First line: "<< str << endl;
  cin.getline(str, sizeof(str));
  cout <<"Second line: "<< str;
}//----------------------
```

运行结果为：

```
Type in a line of text press Enter
You should look "X" up in the subject catalogue.
First line: You should look "
Second line: " up in the subject catalogue.
```

运行时，当出现输入文本的提示时，输入一组以 X 分隔开的单词。当程序显示其输出时，将把 X 以前的文本显示为一行，X 以后的文本显示为一行，不包括 X 字符本身。

程序中的 X 为大小写敏感的。一个小写 x 不会结束第一个 cin.getline()的输入，而且，在输入 X 之前，可以按一到多次回车键，而并不结束第一个 cin.getline ()的输入。第一个 cin.getline()的输入操作将以键入 X 后的第一个回车结束。



→ "cin.getline();"与"cin ≫str;"的一个不同是，前者输入一行，行中可以包含空格，后者却以空格或回车作为字串结束，不包含空格。

2. 用 get()读取一个字符

根据程序的输入要求，有时需要执行每次输入一个字符。这时，可以使用 get()成员函数。其格式为：

```
char istream::get();
```

例如，下面程序循环读入字符，直到用户输入一个 Y 字符，或遇到 ctrl－Z(文件尾)：

```
//---------------------
//     ch19_11.cpp
//---------------------
#include <iostream>
#include <cctype>
using namespace std;
//---------------------
int main(){
char letter;
while(!cin.eof()){
  letter = cin.get();
  letter = toupper(letter);
  if(letter == 'Y'){
    cout <<"'Y' be met.";
    break;
    }
    cout << letter;
  }
}//--------------------
```

如上所示，该程序在遇到一个 Y 字符之前做简单循环，为了简化测试，该程序将每个字符都转换成大写。

"char toupper(char);" 函数原型在 ctype.h 或 cctype 中声明，如果参数为小写字母，将其转换为大写字母，否则，原样返回。上例中函数 toupper()将 letter 转换为大写字母后，赋值给 letter 变量，所以 letter 变量值被改变。

使用流成员函数的输入操作不只限于键盘，上例程序可从重定向输入中每次读入一个字符。下面的命令把 ch19_11.cpp 文件作为重定向输入，并输出运行结果：

```
c >ch19_11 < ch19_11.cpp
 #include < iostream.h>
 #include < cct'Y' be met.
```

→ "letter＝cin.get();"与"cin≫letter;"都是从输入流中取一个字符，但却有区别。默认情况下，cin≫letter 将跳过在文件中发现的任何空白字符(空白字符指空格、tab 符、backspace 符和回车符)。而 cin.get()则不跳过空白字符。

3. 用 get()输入一系列字符

用 get()成员函数的第二种形式可以输入一系列字符，直到输入流中出现结束符或所



读字符个数已达到要求。其原型为：

```
istream& istream::get(char * ,int n,char delim = '\n');
```

由于可以规定输入字符个数,所以下面不安全的代码：

```
ifstream fin("abc.dat");
char buffer[80];
fin >> buffer;          //不能保证输入字符个数在 80 以内
```

可以改写为：

```
istream fin("abc.dat");
char buffer[80];
fin.get(buffer,80);      //保证输入字符个数在 80 以内
```

→ getline()和 get()第二种形式相同。唯一的不同是 getline()从输入流中输入一系列字符时包括分隔符,而 get()不包括分隔符。

4. 用 put()输出一个字符

下例程序使用 put()成员函数,在屏幕上依次输出字母表中的字母：

```
//---------------------
//    ch19_12.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
int main(){
  for(char letter = 'A'; letter <= 'Z'; letter++)
    cout.put(letter);
}//--------------------
```

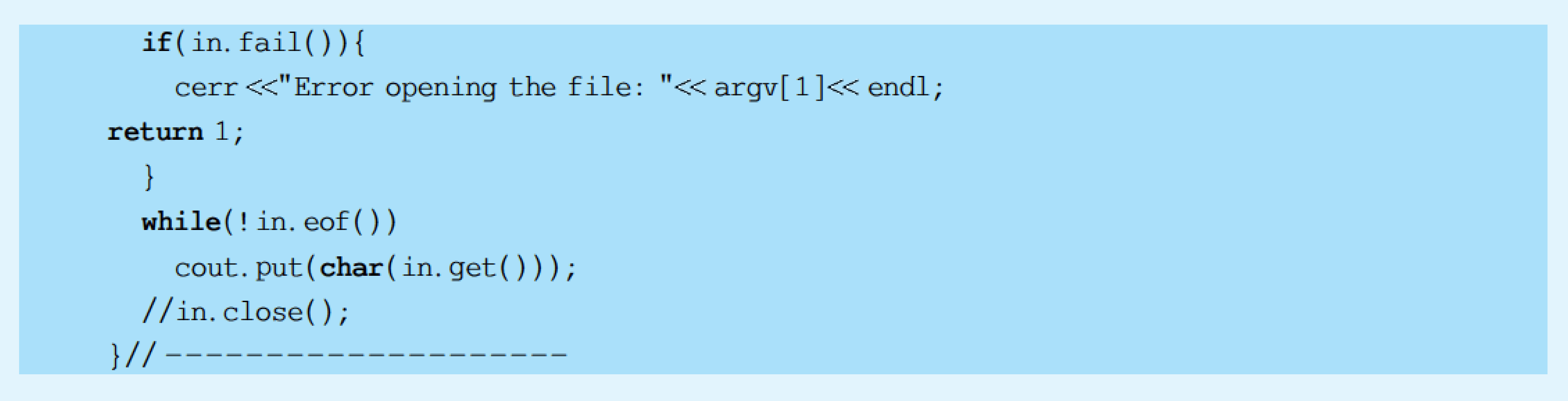
运行结果为：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

→ cout << letter; 与 cout.put(letter); 有一个区别,前者显示以该数据类型表示的形式,后者将参数值以字符方式显示。所以,若 letter 是 char 型,那么这两种方法都可以用来显示字母,若 letter 为 int 型,那么前者将在屏幕上显示 65 到 90 的数字,而不是字母 A 到 Z。

上述流成员函数同样适用于文件流和串流。例如,下面的程序打开以命令行变元形式指定的文件,然后逐个读取字符并显示：

```
//---------------------
//    ch19_13.cpp
//---------------------
#include <fstream>
#include <iostream>
using namespace std;
//---------------------
int main(int argc, char * * argv ){
  ifstream in(argv[1]);
```



```
  if(in.fail()){
    cerr<<"Error opening the file: "<<argv[1]<<endl;
return 1;
  }
  while(!in.eof())
    cout.put(char(in.get()));
  //in.close();
}//--------------------
```

put()成员函数的参数,是文件流对象 in 的成员函数 get()的返回值。

19.7 重载插入运算符

C++重载左移运算符≪执行输出,对程序员来说很方便。因为可以重载同一个运算符去执行自己定义的类输出。左移运算符也称插入运算符,它比较形象,执行"cout≪x;"输出时,好像 x 被插入到输出设备上。重载插入运算符的特性使得流 I/O 可扩展,这与 printf()是重要的区别。

例如,下面程序进行插入运算符重载,打印人民币类对象:

```
//---------------------
//    ch19_14.cpp
//---------------------
#include<iostream>
#include<iomanip>
using namespace std;
//---------------------
class RMB{
  unsigned int yuan;
  unsigned int jf;
public:
  RMB(double v=0.0){
    yuan=v;                //yuan 得到 v 的整数部分
    jf = (v-yuan)*100.0+0.5;
  }
  operator double(){  return yuan+jf/100.0; }
void display(ostream& out){
    out<<yuan<<'.'<<setfill('0')<<setw(2)<<jf   //如:8 分显示 08
       <<setfill(' ');
  }
}; //-------------------
ostream& operator<<(ostream& oo, RMB& d){          //重载插入运算符
  d.display(oo);
  return oo;
}//--------------------
int main(){
  RMBrmb(1.5);
  cout<<"Initially rmb="<<rmb<<"\n";
  rmb=2.0*rmb;
  cout<<"then rmb="<<rmb<<"\n";
}//--------------------
```



运行结果为:

```
Initially rmb = 1.50
then rmb = 3.00
```

display(ostream& out)中的参数是向它传递的任何 ostream 对象输出,并不一定是 cout 输出,这是出于灵活的考虑。它允许 fstream 和 strstream 对象,fstream 操作的对象是设备中的文件,典型的例子就是磁盘文件,strstream 操作的对象是内存中的字符串。因为 fstream 和 strstream 都是 ostream 的子类,这样就便于输出到不同的地方。如定义一个文件流,并将 rmb 对象输出:

```
ofstream fout("abc.dat");
fout << rmb;
```

这时,“fout≪rmb;”仍能够匹配重载的插入运算符。

重载插入运算符 ostream& operator≪() 调用了 RMB 类的成员 display(),以此来完成输出任务,所以它自己不必是友元。这样,RMB 类的界面显得更干净。

由于输出都是通过重载插入运算符来完成,所以也可以更简单地把成员函数 display()里面所做的一切都挪到重载插入运算符里来:

```
ostream& operator <<(ostream& oo, RMB& d){
   oo << d.yuan <<'.'
      << setfill('0') << setw(2) << d.jf
      << setfill(' ');
   return oo;
}
```

这时候,重载插入运算符应为 RMB 类的友元,因为它要直接访问 RMB 类的保护数据。但是它不能是成员,因为首先,插入运算符跟在 ostream 对象的后面,显然它不能是 RMB 类的成员;其次,ostream 类在 iostream 头文件中定义,是标准类库,用户只能继承,不能修改标准类库,所以它更不能是 ostream 类的成员。

→ 重载插入运算符中最后一条语句是“return oo;”,为什么要返回传递给它的 ostream 对象?

这样允许该运算符在单个表达式中与其他插入运算符联结在一起。≪的运算顺序是从左到右,下面的表达式

```
void fn(RMB r, float interest)
{
  cout <<"Sun of these  = "<< r <<" + " << interest << endl;
}
```

被翻译成:

```
void fn(RMB r, float interest)
{
  ((((cout <<"Sum of these  = ")<< r)<<" + ")<< interest)<< endl;
}
```

第一次插入向 cout 输出字符串“Sum of these =”。该表达式的结果是对象 cout,然后它被传递给重载插入运算符

```
ostream& operator <<(ostream&, RMB&)
```

这个运算符返回它的 ostream 对象,这一点很重要,这样做,对象才能被传递给下一个插入运算符。

假设重载插入运算符返回类型是 void,像前面那样相当有效的用法将出现编译错。因为你不能向 void 内插入一个字符串。下面错误更糟糕,因为更难发现:

```
ostream& operator <<(ostream& os, RMB& rmb)
{
  rmb.display(os);
  return cout;
}
```

这个运算符没有返回它给出的 ostream 对象,而返回 ostream 对象 cout。cout 最常用,编译也正确,但是在下面程序中:

```
void fn(int acc, RMB& balance, char * pName)
{
  ofstream outf("accounts.dat", ios::ate);
  outf << acc << balance << pName << endl;
}
```

int acc 通过 outf 的插入运算符输出到 outf 中,返回 outf。然后 RMB 通过重载插入运算符输出到 outf 中,可是错误地返回 cout,而不是 outf,现在 pName 输出到 cout 中,而不是输出到想要输出的文件中。

19.8 插入运算符与虚函数

插入运算符不能是成员函数,也就不能成为虚函数,因此对于上节中的重载插入运算符定义,在下例的派生类中,显得无能为力:

```
class DerivedRMB :public RMB
{
   public:
     DerivedRMB(double v, int n) :RMB(v),c(n){}
   protected:
     int c;
};

DerivedRMB a(5.2,3);

void fn()
{
   cout << a;       //a.c 将不能显示
}
```

cout≪a;能匹配重载的插入运算符,但是执行的结果是输出 a 的人民币值而不能输出 a 作为派生的附加信息 a.c。

解决方法:在重载插入运算符中,不直接实现输出,而是调用 display()成员,再将 display()定义为虚函数。这样,重载插入运算符的行为便可随 display()的不同而不同。这



就是上一节为什么要间接实现重载插入运算符的用意。

1. 先定义一个抽象类

```
class Currency
{
   public:
     Currency(double v  = 0.0)
     {
        yuan = v;
        jf = (v - yuan) * 100.0 + 0.5;
     }

     virtual void display(ostream& out) = 0;
   protected:
     unsigned int yuan;
     unsigned int jf;
};
```

2. 派生人民币类等子类

```
class RMB :public Currency
{
   public:
     RMB(double v  = 0.0) :Currency(v){}
     virtual void display(ostream& out)
     {
        out << yuan <<'.'
             << setfill('0') << setw(2) << jf << setfill(' ')
             <<" RMB";
     }
};

class EUR :public Currency      //欧元
{
   EUR(double v = 0.0):Currency(v){}
   virtual void display(ostream& out)
   {
     out << yuan <<'.'
          << setfill('0') << setw(2) << jf << setfill(' ');
     out <<" EUR";
   }
};

class USD :public Currency      //美元
{
   public:
     USD(double v = 0.0):Currency(v){}
     virtual void display(ostream& out)
     {
```

```
        RMB::display(out);
        out <<" USD";
      }
  };
```

3. 定义插入运算符

```
ostream& operator <<(ostream& oo, Currency& c)
{
   c.display(oo);
   oo << endl;
   return oo;
}
```

4. 应用

```
void fn(Currency& c)
{
   cout <<"Deposit is " << c << endl;
}

int main()
{
  RMB rmb(5.5);
  fn(rmb);
  USD usd(5.6);
  fn(usd);
  EUR eur(10.6);
  fn(eur);
}
```

运行结果为：

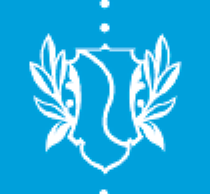
```
Deposit is 5.5 RMB
Deposit is 5.6 USD
deposit is 10.6 EUR
```

类 Currency 有三个子类 RMB、EUR 和 USD。Currency 中 display()成员定义为纯虚函数。在每一个子类中，display()成员被重载，从而可以适当的格式输出相应对象。重载插入运算符函数对 display()的调用是一个虚调用。因此当它被传递以 RMB 类对象时，则像人民币那样输出；当它被传递以 EUR 对象时，则像欧元那样输出。因而，尽管重载的插入运算符不是虚拟的，因为它调用了一个虚函数，结果令人满意。

重载插入运算符的参数 Currency& 必须为引用，如果以值传递，那么对于：

```
ostream& operator <<(ostream& oo, Currency& c)
```

编译将报错。因为 Currency 是抽象类，不能构造该类的对象。



19.9 文件操作

1. 文件输出

在文件输出时,往往涉及大量相关记录的集合。把文件看作是相关记录的集合。如要输出若干个学生对象,每个学生对象含有学生姓名、学号、成绩等,将其看作是一个数据记录。

例如,下面的程序输出 3 个学生对象,其中一个大学生,两个硕士生。程序由一个工程文件 ch19_15.prj 组成:

```
//*********************
// ch19_15.prj
//*********************
ch19_15.cpp
student.cpp
master.cpp
//*********************
```

其头文件和程序源文件分别为:

```
//---------------------
//student.h
//---------------------
#ifndef STUDENT
#define STUDENT
#include <iostream>
#include <cstring>
using namespace std;
//---------------------
class Student{                                   //大学生类
  char pName[20];
  unsigned int uID;
  float grade;
public:
  Student(char* pS, unsigned num, float g){
    strcpy(pName,pS);
    uID = num;
    grade = g;
  }
  virtual void display(ostream& out);
}; //-------------------
ostream& operator<<(ostream& out, Student& st);
#endif

//---------------------
//     student.cpp
//---------------------
#include"student.h"
#include <iomanip>
#include <iostream>
using namespace std;
```

```
//---------------------
void Student::display(ostream& out){
  out << left << setw(20)<< pName << uID <<","
     << right << setw(4)<< grade;
}//---------------------
ostream& operator <<(ostream& out, Student& st){   //插入操作符
  st.display(out);
  out << endl;
  return out;
}//---------------------

//---------------------
//master.h
//---------------------
#include"student.h"
#include <iostream>
using namespace std;
//---------------------
class MasterStudent: public Student{              //硕士生类
  char type;
public:
  MasterStudent(char * pS, unsigned num, float g, char t)
  :Student(pS,num,g),type(t){}
  void display(ostream& out);
}; //---------------------

//---------------------
//    master.cpp
//---------------------
#include"master.h"
#include <iostream.h>
using namespace std;
//---------------------
void MasterStudent::display(ostream& out){
  Student::display(out);
  out <<", "<< type;
}//---------------------

//---------------------
//    ch19_15.cpp
//---------------------
#include"student.h"
#include"master.h"
#include <fstream.h>
using namespace std;
//---------------------
int main(){
  ofstream out("e:\\bctemp\\abc.txt");
  Students1("Dill Arnson", 12567, 3.5);
  MasterStudent s2("Welch Shammas", 12667, 4.1, 'A');
  MasterStudent s3("Portel Braumbel", 12579, 3.8, 'B');
  out << s1;
  out << s2;
  out << s3;
}//---------------------
```



运行结果可以在打开的文件中看到:

```
e:\> type abc.txt
Dill Arnson          12567, 3.5
Welch Shammas        12667, 4.1, A
Portel Braumbel      12579, 3.8, B
```

2. 文件输入

如果要打开一个文件用于输入,可以用 ifstream 类。用上例中的类定义,假定文件 abc.txt 中的内容为:

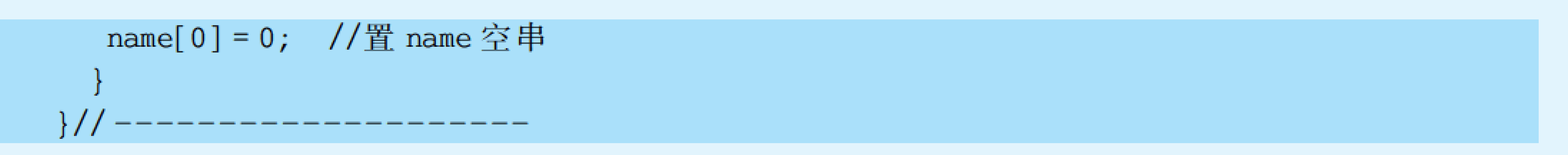
```
Dill Arnson        12567 3.5 A
Welch Shammas      12667 4.1 A
Portel Braumbel    12579 3.8 B
```

将该文件的内容逐条输入,创建 MasterStudent 对象,屏幕输出该记录内容。其程序为一个工程:

```
//*********************
// ch19_16.prj
//*********************
ch19_16.cpp
student.cpp
master.cpp
//*********************
```

其 ch19_16.cpp 文件内容为:

```
//---------------------
//    ch19_16.cpp
//---------------------
#include"student.h"
#include"master.h"
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;
//---------------------
int main(){
  ifstream fin("e:\\bctemp\\abc.txt");
  char sFirst[10];
  char sLast[10];
  unsigned int uid;
  float nGrade;
  char type;
  char name[20];
  Student* pS;
  for(int i=0; fin>>sLast>>sFirst>>uid>>nGrade>>type;){
  strcpy(name,strcat(sLast," "));
  strcpy(name,strcat(name,sFirst));
  pS = new MasterStudent(name,uid,nGrade,type);
  cout<<"student #"<<++i<<":"<<*pS;
  delete pS;
```



```
    name[0] = 0;  //置 name 空串
  }
}//--------------------
```

运行结果为：

```
student #1:Dill Arnson        12567, 3.5, A
student #2:Welch Shammas      12667, 4.1, A
student #3:Portel Braumbel    12579, 3.8, B
```

程序中先将文件的一行分 5 个数据项读入(一条记录)，然后将前两项组合成一个名字。用名字 name、学号 uid、成绩 nGrade 和类别 type 这四项初始化一个 MasterStudent 堆对象，在屏幕上输出。随后，继续这一过程，直到文件尾。文件尾的判断标志为 fin.eof()成员函数，1 表示文件尾，0 表示未到文件尾。

小结

C 的 I/O 是丰富、灵活和强大的，但是，C 的 I/O 系统一点也不了解对象，不具有类型的安全性。C++的 I/O 流扬弃了 C 的 I/O 系统，它操作更简捷、更易理解，它使标准 I/O 流、文件流和 C 字串流的操作在概念上统一了起来。有了控制符，C++更灵活。由它所重载的插入运算符完全融入了 C++的类及其继承的体系。

本章介绍了文件操作的文本顺序读写。文件的其他操作一样可以完成，进一步的学习请看其他参考书。

练习

19.1 编写一个程序，它读入一个文件，并统计文件中的行数。

19.2 改写第 19.7 节中 RMB 类的重载插入运算符，使得人民币输出格式为：长度一律 10 位，小数两位，币值前有一¥符号。

19.3 设字符串 string ="1 2 3 4 5 6 7 8 9 "，用串流 I/O 的方法编程逐个读取这个串的每个数，直到读完为止，并在屏幕上输出。

19.4 设置格式，显示 ABCDE…Z 这 26 个字母的 ASCII 码大写字母十六进制数。

19.5 实现一个通讯录打印程序。通讯录的记录格式为：

姓名，单位，电话，住址，宅电

要求先建立 Person 类，然后按对象读入和打印。

第20章 模板

模板是 C++语言相对较新的一个重要特性。模板使程序员能够快速建立具有类型安全的类库集合和函数集合，它的实现方便了更大规模的软件开发。本章介绍了模板的概念、定义和使用模板的方法，通过这些介绍，使读者有效地把握模板，以便能正确使用 C++系统中日渐庞大的标准模板类库。

20.1 模板的概念

若一个程序的功能是对某种特定的数据类型进行处理，则若将所处理的数据类型说明为参数，就可把这个程序改写为模板。模板可以让程序对任何其他数据类型进行同样方式的处理。

C++程序由类和函数组成，模板也分为类模板(class template)和函数模板(function template)。因此，可以使用一个带多种不同数据类型的函数和类，而不必显式记忆针对不同的数据类型的各种具体版本。

函数模板的一般定义形式是：

```
template <类型形式参数表> 返回类型 FunctionName ( 形式参数表 )
{
   //函数定义体
}
```

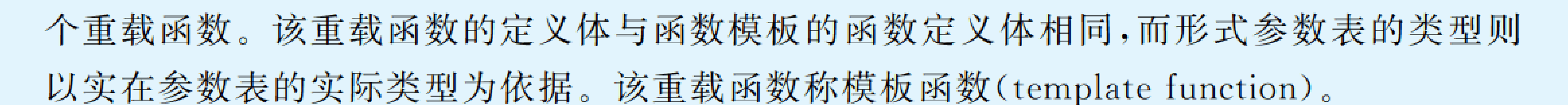
其中的类型形式参数表可以包含基本数据类型，也可以包含类类型。如果是类类型，则须加前缀 class。

这样的函数模板定义，不是一个实实在在的函数，编译系统不为其产生任何执行代码。该定义只是对函数的描述，表示它每次能单独处理在类型形式参数表中说明的数据类型。

当编译系统发现有一个函数调用：

```
FunctionName ( 实在参数表 );
```

将根据实在参数表中的类型，确认是否匹配函数模板中对应的形式参数表，然后生成一


个重载函数。该重载函数的定义体与函数模板的函数定义体相同，而形式参数表的类型则以实在参数表的实际类型为依据。该重载函数称模板函数(template function)。

→ 函数模板与模板函数的区别

函数模板是模板的定义，定义中使用通用类型参数。

模板函数是实实在在的函数定义，它是由函数模板生成的。编译系统在发现具体的函数调用时，匹配类型参数，生成函数代码。

类模板的一般说明形式是：

```
template <类型形式参数表> class className
{
   //类声明体
};

template <类型形式参数表>
返回类型 className <类型名表>::MemberFuncName1 ( 形式参数表 )
{
   //成员函数定义体
}

template <类型形式参数表>
返回类型 className <类型名表>::MemberFuncName2 ( 形式参数表 )
{
   //成员函数定义体
}

…

template <类型形式参数表>
返回类型 className <类型名表>::MemberFuncNameN ( 形式参数表 )
{
   //成员函数定义体
}
```

其中的类型形式参数表与函数模板中的意义一样。后面的成员函数定义中，className <类型名表>中的类型名表，是类型形式参数的使用。

这样的一个说明(包括成员函数模板定义)，不是一个实实在在的类，只是对类的描述，称为类模板(class template)。

建立类模板之后，可用下列方式创建类模板的实例：

```
className <类型实在参数表> object;
```

其中类型实在参数表应与该类模板中的类型形式参数表匹配。class_name <类型实在参数表>是模板类(template class)，object 是该模板类的一个对象。

→ 类模板与模板类的区别。

类模板是模板的定义，定义中使用通用类型参数。

模板类是实实在在的类定义，是由类模板生成的。编译系统在发现以类模板方式创建其实例(对象)时，匹配类型参数，生成模板类定义。该模板类创建的对象即类模板的实例。



20.2 为什么要用模板

1. 关于函数

考察两个 swap()函数,一个交换两个整型数,另一个交换两个浮点数。两个 swap ()的主体行为是一样的,一个处理 int 型,另一个处理 float 型。两个函数分别定义如下:

```
void swap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

swap(float& a, float& b)
{
    float temp = a;
    a = b;
    b = temp;
}
```

交换任何一对类类型对象,可以定义如下:

```
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

能不能对于任一类型 T 的两个对象 x1 和 x2,函数调用 swap(x1,x2)总能使编译系统理解其交换意义而予以实现呢?若不然,每交换一对新类型的对象,都要定义一个执行同样操作的重载函数。

有了函数模板之后,重载就不必要了。

2. 关于类

再来考察一个链表类。对于 Cat 类的链表,我们有:

```
class Cat
{
    //...
};

class CatList
{
    public:
      Cat List();
      void Add(Cat&);
      void Remove(Cat&);
      Cat * Find(Cat&);
      ~Cat List();
```

```
    private:
      Cat * first;
      //...
};
```

该链表类将 Cat 类对象作为链表结点,进行插入、删除和查找处理,所以称之为 CatList 类。

同样如果想处理其他的任何一种类类型的对象作为结点的链表,我们必须重新对这链表进行定义。这种工作很烦琐,因为类的行为没有任何变化,只是处理的结点之类型有所不同。

如果让类模板来工作,就能最大限度地解决这些问题。

20.3 函数模板

对于具有各种参数类型,相同个数、相同顺序的同一函数(重载函数),如果用宏定义来写:

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

则它不能检查其数据类型,损害了类型安全性。这也是为什么要使用函数模板的一个原因。

另外,如果一个个地定义其函数重载则不简洁,例如,求同一数据类型数值中的最大值:

```
int max(int a, int b)
{
   return a>b?a:b;
}

float max(float a, float b)
{
   return a>b?a:b;
}
```

对于与整数相容的 char 类型数据值的调用,也不能得到满意的结果。如调用:

```
max('3', '5');
```

则编译系统会为其找到一个 int 型的匹配,调用“int max(int a,int b);”函数,但是它将返回 53 而不是'5'(其 ASCII 码是 53)。

用函数模板可将许多重载函数简单地归为一个,如下例所示:

```
//---------------------
//     ch20_1.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
template<class T>
Tmax(T a,T b){
  return a > b ? a : b;        //T类的>操作须有定义
}//--------------------
int main(){
  cout <<"Max(3,5) is "<< max(3,5)<< endl;
```




```
    cout <<"Max('3','5') is "<< max('3','5')<< endl;
}//---------------------
```

运行结果为:

```
Max(3,5) is 5
Max('3','5') is 5
```

当编译发现用指定数据类型调用函数模板时,就创建一个模板函数。

上例中,当编译程序发现 max(3,5)调用时,它就产生了一个如下的函数定义,生成其程序代码:

```
int max(int a,int b)
{
    return a > b?a:b;
}
```

当发现 max('3','5')调用时,它又产生另一个如下的函数定义,也生成其程序代码:

```
char max(char a, char b)
{
    return a > b?a:b;
}
```

这样,实参是什么数据类型,返回值也是什么数据类型,不会出现前面的问题。而且模板又避免了相同操作的重载函数定义。

20.4 重载模板函数

可以像重载普通函数那样重载模板函数。

```
//---------------------
//    ch20_2.cpp
//---------------------
#include <iostream>
#include <cstring>
using namespace std;
//---------------------
template<class T> T max(T a,T b){
  return a>b ? a : b;
}//---------------------
char* max(char* a, char* b){
  return strcmp(a,b)>= 0 ? a : b;
}//---------------------
int main(){
  cout <<"Max(\"Hello\",\"Gold\") is "
      << max("Hello","Gold")<< endl;
}//---------------------
```

运行结果为:

```
Max("Hello","Gold") is Hello
```

函数 char * max(char * ,char *)中的名字 max 与函数模板的名字相同,但操作不同,



函数体中的比较采用了字符串比较函数,所以有必要用重载的方法把它们区分开,这种情况就是重载模板函数。编译程序在处理这种情况时,首先匹配重载函数,然后再寻求模板的匹配。

编译程序看到 max("Hello","Gold")调用时,先进行重载函数匹配,结果匹配了非模板函数 char * max(char * ,char *),所以这里不会为它产生模板函数的代码。

20.5 类模板的定义

链表操作并不依赖于要处理的链表的数据类型(如在上面所说的 Cat 类)。定义类模板时,就是利用了这种独立性。链表操作时,只是把要处理的数据类型当作参数。一个类模板用于构筑一个通用链表,如整数链表、结构链表,以及任何其他定义过的数据类型的链表。上节描述的 CatList 类也可以通过通用链表来构筑。

用类模板来定义一个通用链表,此时该通用链表还不是一个类定义,只是类定义的一个框架,即类模板。

下例定义了一个单向链表的类模板,它分别实现增加、删除、查询和打印操作,见图 20-1。

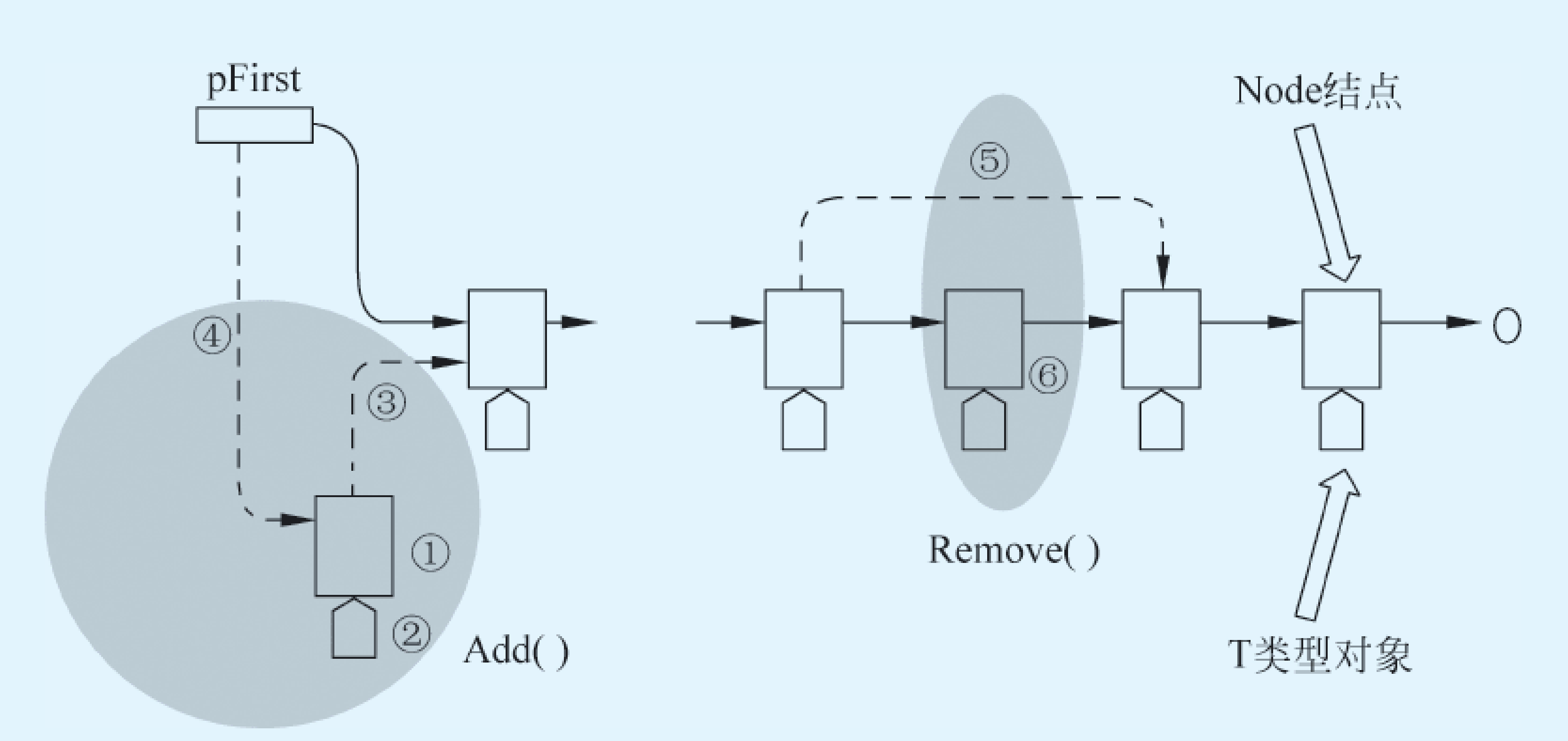


图 20-1 单向通用链表操作示意图

增加时,Add()成员函数在链首挂接上一个携有 T 类型对象的结点,使之成为链首结点,具体由下列步骤①、②、③和④完成:

① 从堆空间申请一个结点;

② 将 T 类对象挂接在这个结点上;

③ 将该结点指向链首的结点;

④ 将该结点成为链首(链首指针 pFirst 指向它)。

删除时,Remove()成员函数通过调用 Find()成员函数寻找到挂接该对象的结点,先脱链,再删除结点。由下列步骤⑤和⑥完成:

⑤ 链中待删结点前后的两个结点链接起来,以使待删结点脱链;

⑥ 删除待删结点(将空间还给堆)。

如果找不到对应结点,则无功而返;如果找到的是链首结点,则步骤⑤的脱链,由链首指针 pFirst 指向下一个结点来完成。



查询时,Find()成员函数从链首开始遍历整个链表,查找是否有结点含有对应对象,若无,则返回空指针。

打印时,PrintList()成员函数从链首开始遍历整个链表,打印每个结点下的对象值,因而要求 T 类型的输出运算符须有定义。

链表类包含唯一的一个数据成员 pFirst,刚创建对象时,pFirst 指向空,链表也为空。当进行增加操作后,链表非空,而 pFirst 指向链首。所有成员函数的操作,都从 pFirst 开始。

链表类析构的任务是把链表中所有的结点空间收回,它从链首开始遍历整个链表,先删除一个结点,然后去处理下一个结点,逐个推进。代码如下:

```
//---------------------
//listtmp.h
//---------------------
#ifndef LIST
#define LIST
#include<iostream>
using namespace std;
//---------------------
template<class T> struct Node{
  Node* pNext;
  T tValue;
}; //-------------------
template<class T> class List{
  Node<T>* pFirst, *pivot;        //链首结点指针,哨兵指针⇙
public:
  List(){ pFirst=0; }
  void Add(T&);
  void Remove(T&);
  Node<T>* Find(T&);
  void PrintList();
~List();
}; //-------------------
template<class T>
void List<T>::Add(T& t){
  Node<T>* tmp=new Node<T>;       //①
  tmp->tValue=t;                  //②
  tmp->pNext=pFirst;              //③
  pFirst=tmp;                     //④
}//--------------------
template<class T>
void List<T>::Remove(T& t){
  Node<T>* q = Find(t);           //定位待删的结点
  if(!q) return;
  if(q==pFirst)
    pFirst=pFirst->pNext;         //⑤
  else
    pivot->pNext=q->pNext;        //⑤
  delete q;                       //⑥
}//--------------------
template<class T>
Node<T>* List<T>::Find(T& t){
```

```
    if(pFirst->tValue==t)
      return pFirst;
    for(Node<T>* p=pFirst->pNext; p; pivot=p,p=p->pNext)
      if(p->tValue==t)
        return p;              // pivot 指向上一结点
    return 0;
}//--------------------
template<class T>
void List<T>::PrintList(){
  for(Node<T>* p=pFirst; p; p=p->pNext)
    cout<<p->tValue<<" ";     //须有 T 的友元处理 T 对象输出
  cout<<endl;
}//--------------------
template<class T>
List<T>::~List(){
  for(Node<T>* p; p=pFirst; delete p)
    pFirst=pFirst->pNext;
}//--------------------
#endif
}//--------------------
```

上例类模板中，用到了一个 Node 结构模板。Node 结构变量是通用链表类中的链表结点，只在通用链表类的范围内操作，所以，该定义放在类模板定义的头文件中。

20.6 使用类模板

使用类模板的方法为：

(1) 在程序开始的头文件中说明类模板的定义。

(2) 在适当的地方创建一个类模板的实例，编译发现正在创建一个模板类的对象时，便会根据类模板生成模板类的定义，同时，创建相应的对象实体。

(3) 有了对象名，以后的使用就和通常一样。但是规定了什么类型的模板类，在调用成员函数时，所赋的实参也要对应该类型。

例如：

```
//---------------------
//    ch20_3.cpp
//---------------------
#include"listtmp.h"
//---------------------
int main(){
  List<float> floatList;
  for(int i=1; i<7; i++)
    floatList.Add(*new float(i+0.6));

  floatList.PrintList();
  float b=3.6;
  floatList.Remove(b);
  floatList.PrintList();
}//--------------------
```



运行结果为：

```
6.6 5.6 4.6 3.6 2.6 1.6
6.6 5.6 4.6 2.6 1.6
```

上例类模板定义(包含其成员函数模板定义)都在头文件中，这与一般类定义的方法不同。一般类定义时，类定义部分作为界面放在头文件中，成员函数定义部分作为实现放在cpp文件中。但作为模板，在应用程序中，编译发现 List < float > floatList；这样的声明时，要为其生成模板类的实实在在的定义，所以模板中必须包含整个模板(包含其成员函数)的完整定义，在生成模板类之后，系统又为其创建该类的对象。

创建了模板类对象，就是创建了一个链表，这时链表为空。

程序执行了 Add() 操作后，链表便一个个连起来。执行 Add()时，程序先从堆中申请分配 float 变量空间，初始化，然后作为参数传递给 Add()，Add() 马上从堆中申请分配结点空间，挂接该 float 变量，最后作为链首链入链表中。

程序创建了 float 变量并赋值，然后以此作为实际参数，调用 Remove()成员函数。

20.7 使用标准模板类库：Josephus 问题

标准模板类库 STL(Standard Template Library)是一个基于模板的容器类库，它包括向量、链表、队列和栈，还包括了一些通用的算法，如排序和查找等。它已经成为 C++标准的组成部分。使用标准模板类库的好处是：可以避免自己在开发模板类库时，不同模板类之间的功能重复；最大限度的类库重用；作为 C++标准，可移植性强是不言而喻的。面向对象程序设计和面向对象数据库设计时，大量用到容器类库，所以，标准模板类库对 C++的发展来说，影响将是巨大的。

我们使用 BCB6.0 中提供的标准模板类库，用其中的 list 链表类模板来求解 Josephus 问题。

该模板类库 list 链表是双向链表，所以性能比单向链表要高。其模板类名为< T >，其中 T 表示由链表处理的对象类型，用到的有：

insert(结点地址，T&)成员函数，在结点地址前插入。

erase(结点地址)成员函数，删除结点后，返回下一个结点地址。

链表迭代算子 list < T >::iterator 作为友类，其对象取值为链表中结点的地址。

链表模板类的操作很像上例的通用链表，只不过模板类名是标准模板类名，使用时，只要声明该标准模板类名，创建其对象，了解并操作其成员函数。成员函数的操作示意见图 20-2。

对应的迭代算子模板类名为 list < T >::iterator，仅用到增量 Object& operator++()操作。其取值包括链表的起始和尾后地址。所谓起始地址即链表第一个结点的地址，所谓尾后地址，即链表尾结点后面的地址，它应该形容成空结点。

在程序中，有三个函数，Init、Step 和 Detach 函数，它们组合了一些模板成员操作，或者反复调用模板成员函数，以达到解决 josephus 问题的目的。

Init 函数调用模板类中的 insert 成员函数，其参数为插入地址和结点值。结点插入在该参数地址之前，反复以尾后地址作为参数，其效果是链表按插入顺序链接各结点。

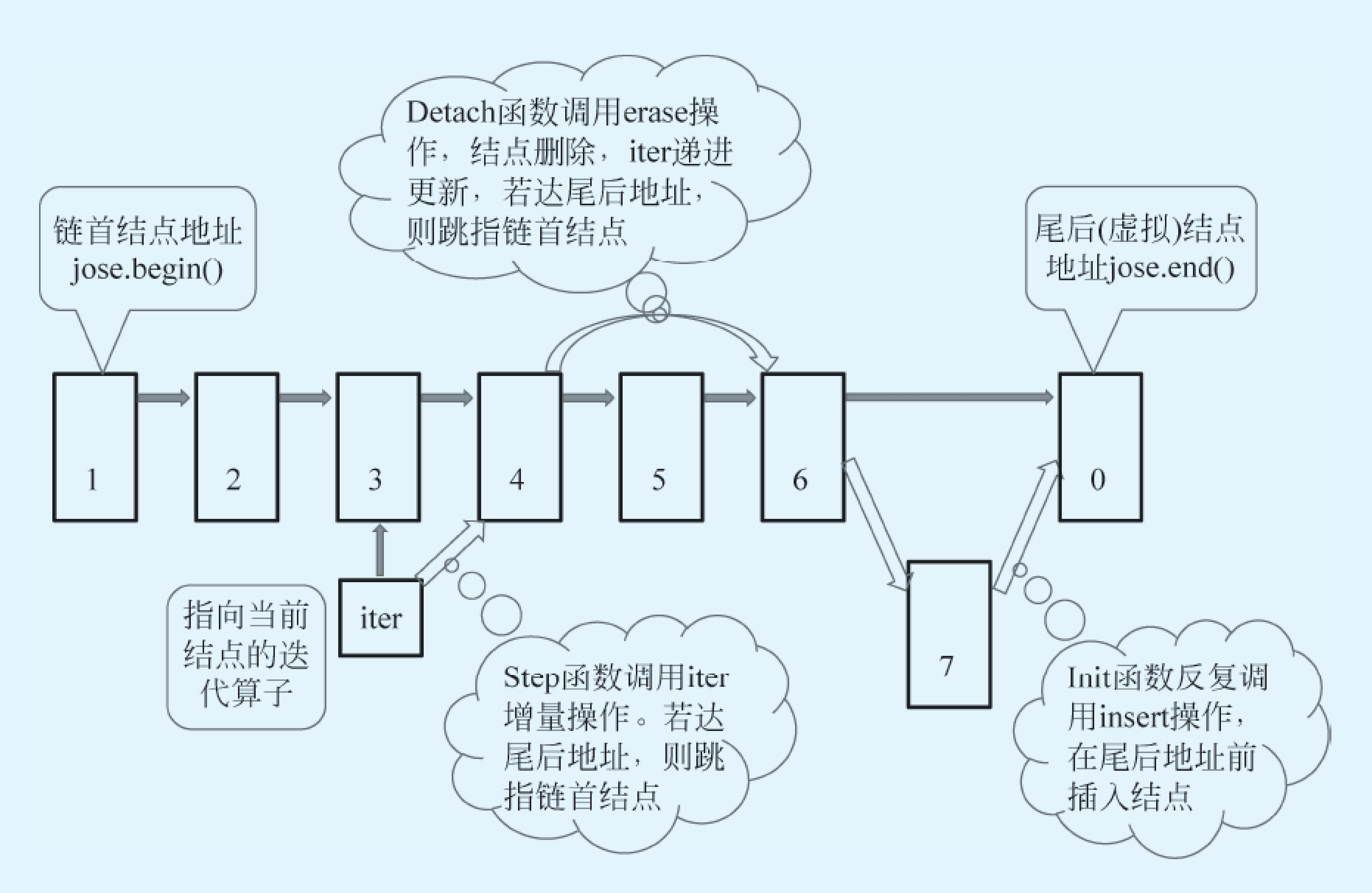


图 20-2　单向链表模板类操作示意

Step 函数则反复调用迭代算子的增量操作，为了实现环链表的功能，增量的同时，须判断是否越过链尾到达尾后地址，若是，则调整其迭代算子为链首地址。

至于 Detach 函数，则将当前迭代算子指向的结点脱链，并保证迭代算子指向的是下一个结点。因此，判断越过链尾到达尾后地址是必需的。

标准模板类和迭代算子有许多有用的成员函数，这里没有用到的都省略。

下例是使用标准模板类库的一个 Josephus 问题解法：

```
//----------------------
//     Josephus 问题解法六
//     jose6.cpp
//----------------------
#include <iostream>
#include <iomanip>
#include <list>
using namespace std;
//----------------------
void Detach();                    //小孩离队，置下个小孩为当前位
void Step(int m);                 //指针从下个小孩为当前位起挪 m-1 个位置
void Init(int);                   //初始化小孩编号并输出
//----------------------
list<int> jose;                   //创建单向链表模板类的全局对象
list<int>::iterator iter;         //创建 jose 迭代算子(链表结点指针)
//----------------------
  int main(){
  int n=10, s=4, m=4;             //取三个样本数据分别表示小孩数，开数位，数个数
  Init(n);
  iter=jose.begin();
  Step(s+1);                      //到超前一个位置，即下一轮数第一个位置
  for(int i=1; i<n; i++){
    Step(m);
```




```
      cout <<" "<< * iter;
      Detach();
    }
    cout <<"\nThe winner is "<< * iter <<"\n";
}//--------------------
void Init(int n){
  for(int i = 1; i <= n; i++){       //顺序插入并输出
    jose.insert(jose.end(),i);
    cout <<" "<< i;
  }
  cout <<"\n";
}//--------------------
void Detach(){
  iter = jose.erase(iter);           //脱钩 iter 结点,返回下一个结点地址
  if(iter == jose.end())             //若已到尾,则跳到链首,履行环链功能
    iter = jose.begin();
}//--------------------
  void Step(int m){
  for(int i = 1; i < m; i++)
    if(++iter == jose.end())         //环链操作
  iter = jose.begin();
}//--------------------
```

运行结果为：

```
1  2  3  4  5  6  7  8  9  10
8  2  6  1  7  4  3  5  10
The winner is 9
```

小结

模板是一种安全的、高效的重用代码的方式。它被用于参数化类型，在创建对象或函数时所传递的类型参数可以改变其行为。

每个模板类的实例是一个实际的对象，可以像其他对象一样使用，甚至可以作为函数的参数，或作为返回值。

与类和函数的定义不同，类模板和函数模板的定义一般放在头文件中。

在 C++ 中，一个发展趋势是使用标准模板类库(STL)，VC 和 BC 都把它作为编译器的一部分。STL 是一个基于模板的包容类库，包括向量、链表和队列，还包括一些通用的排序和查找算法等。

STL 的目的是为替代那些需要重复编写的通用程序。当理解了如何使用一个 STL 类之后，在所有的程序中不用重新编写就可以使用它。

练习

20.1　编写一个函数模板，它返回两个值中的最小者。但同时要确保能正确处理字符串。

20.2　以下是一个整数栈类的定义：

```
const int SIZE = 100;
class Stack
{
   public:
     Stack();
     ~Stack();
     void Push(int n);
     int Pop();
   private:
     int stack[SIZE];
     int tos;
};
```

编写一个栈的类模板(包括其成员函数定义),以便为任何类型的对象提供栈结构数据操作。

在应用程序中创建整数栈、字符栈和浮点数栈,并提供一些数据供进栈、退栈和打印操作。

第21章 异常处理

在编写程序时，需要尝试确定程序可能出现的错误，然后加入处理错误的代码。例如，当程序执行文件 I/O 操作时，应测试文件打开以及读写操作是否成功，并且在出现错误时做出正确的反应。随着程序复杂性的增加，为处理错误而必须在程序中加入的代码的复杂性也增加了。为使程序更易于测试和处理错误，C++实现了异常处理机制。本章介绍了C++异常处理。程序使用 try、throw 和 catch 语句来支持异常处理。

21.1 异常的概念

在大型软件开发中，最大的问题就是错误连篇的、不稳定的代码。而在设计与实现中，最大的开销是花在测试、查找和修改错误上。

程序的错误，一种是编译错误，即语法错误。如果使用了错误的语法、函数、结构和类，程序就无法被生成运行代码；另一种是在运行时发生的错误，它分为不可预料的逻辑错误和可以预料的运行异常。

逻辑错误是由于不当的设计造成的，如，某个排序算法不合适，导致在边界条件下，不能正常完成排序任务。一般只有当用户做了某些出乎意料的事才会出现逻辑错误，这些错误，安静地潜伏着，连许多大型的优秀软件都不能避免。就像大战之后残留的地雷，在“一切正常”中，突然某人进入了误区，程序发生了“爆炸”。一旦发现了逻辑错误，专门为其写一段处理错误的代码，就可避免错误的发生，比如数组下标溢出检查，这样错误就防范在先了。

运行异常，可以预料，但不能避免，它是由系统运行环境造成的。如，内存空间不足，程序运行中提出内存分配申请得不到满足，就会发生异常；在硬盘上的文件被挪离，或者光盘没有放好，导致程序运行中文件打不开而发生异常；程序中发生除 0 的代码，导致系统除 0 中断；打印机未打开，调制解调器掉线等，导致程序运行中挂接这些设备失败，等等。这些错误会使程序变得脆弱。然而这些错误是能够预料的，通常加入一些预防代码便可防止这些异常。如，对文件打不开时的保护：

```
#include<fstream>
using namespace std;
//...
void f(char* str)
{
  ifstream source(str);           //打开 str 串中的文件
  if(source.fail())               //打不开
  {
    cerr <<"Error opening the file: "<< str << endl;
    exit(1);                      //退出程序
  }
  //...
}
```

异常是一种程序定义的错误，它对程序的逻辑错误进行设防，对运行异常加以控制。C++中，异常是对所能预料的运行错误进行处理的一套实现机制。

21.2 异常的基本思想

在小型程序中，一旦发生异常，一般是将程序立即中断运行，从而无条件释放所有资源。对于大型程序来说，运行中一旦发生异常，应该允许恢复和继续运行。恢复的过程就是把产生异常所造成的恶劣影响去掉，中间可能要涉及一系列的函数调用链的退栈，对象的析构，资源的释放等。继续运行就是异常处理之后，再紧接着异常处理的代码区域中继续运行。在 C++中，异常是指从发生问题的代码区域传递到处理问题的代码区域的一个对象，见图 21-1。

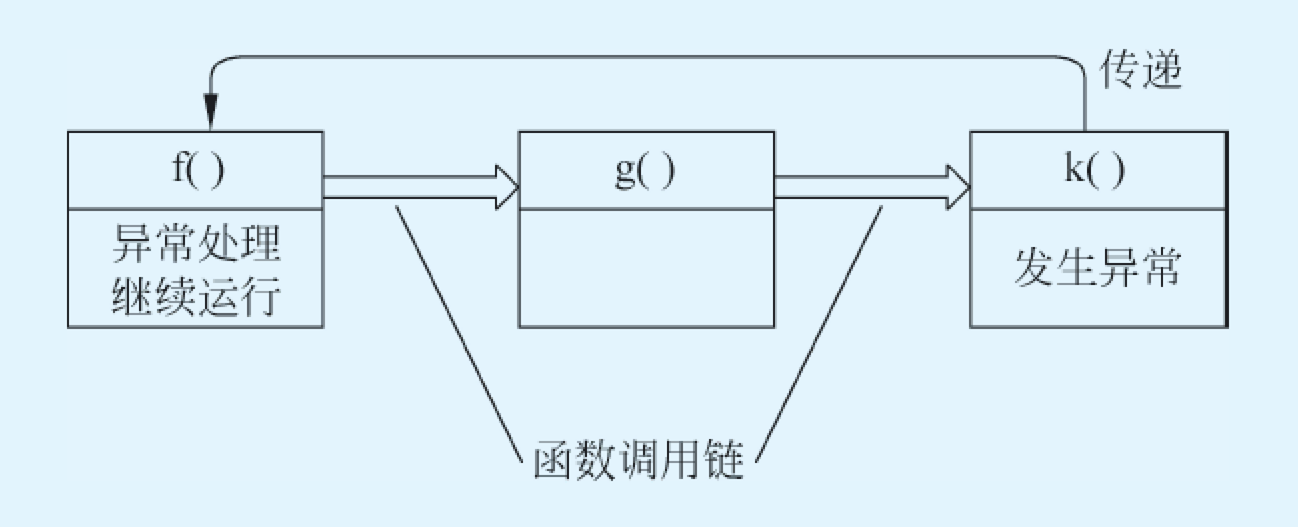


图 21-1　异常的发生、传递与处理

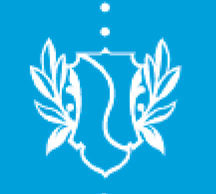
发生异常的地方在函数 k()中，处理异常的地方在其上层函数 f()中，处理异常后，函数 k()和 g()都退栈，然后程序在函数 f()中继续运行。如果不用异常处理机制，在程序中单纯地嵌入错误处理语句，要实现这一目的是艰难的。

异常的基本思想是：

(1) 实际的资源分配(如内存申请或文件打开)通常在程序的低层进行，如图 21-1 中的 k()。

(2) 当操作失败、无法分配内存或无法打开一个文件时，在逻辑上如何进行处理通常是在程序的高层，如图 21-1 中的 f()，处理中间还可能有与用户的对话。

(3) 异常为从分配资源的代码转向处理错误状态的代码提供了一种表达方式。如果还



存在中间层次的函数,如图 21-1 中的 g(),则为它们释放所分配的内存提供了机会,但这并不包括用于传递错误状态信息的代码。

从中可以看出,C++异常处理的目的,是在异常发生时,尽可能地减小破坏,周密地善后,而不去影响其他部分程序的运行。这在大型程序中是非常必要的。对如图 21-1 所示的程序调用关系,如果像上一节处理文件打开失败异常的方法,那么,异常只能在发生的函数 k()中进行处理,无法直接传递到函数 f()中,而且调用链中的函数 g()的善后处理也十分困难。

21.3 异常的实现

使用异常的步骤是:

(1) 定义异常范围(try 语句块)。将那些有可能产生错误的语句框定在 try 块中。

(2) 定义异常处理(catch 语句块)。将异常处理的语句放在 catch 块中,以便异常被传递过来时就处理它。

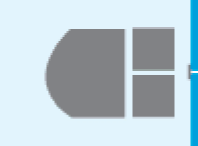
(3) 抛掷异常(throw 语句)。检测是否产生异常,若是,则抛掷异常。

例如,下面的程序,设置了防备文件打不开的异常:

```
//---------------------
//    ch21_1.cpp
//---------------------
#include <fstream>
#include <iostream>
#include <cstdlib>
using namespace std;
//---------------------
int main(int argc, char ** argv){
  ifstream source(argv[1]);      //打开文件
  char line[128];
  try{
    if(source.fail())
      throw argv[1];
  }catch(char * s){
    cout <<"error opening the file "<< s << endl;
    exit(1);
  }
  while(!source.eof()){
    source.getline(line, sizeof(line));
    cout << line << endl;
  }
}//---------------------
```

假定 C 盘中无 abc.txt 文件,有 xyz.txt 文件,内容为 Hello! How are you? 两行问候语句,则运行结果为:

```
c:\> ch21_1 abc.txt
error opening the file abc.txt
c:\> ch21_1 xyz.txt
hello!
How are you?
```



这里抛掷异常(throw argv[1])与处理异常(catch 块)在同一个函数中。当打开文件失败时,就执行"throw argv[1];" 语句,throw 后面的表达式 argv[1]的类型被称为所引发的异常之类型。

try 块结构表示块中的语句可能会发生异常,放在其中加以监控。**C++只理会受监控的过程的异常。**

在 try 块之后必须紧跟一个或多个 catch()语句,目的是对发生的异常进行处理。catch()括号中的声明只能容纳一个形参,当类型与抛掷异常的类型匹配时,该 catch()块便称捕获了一个异常而转到其块中进行异常处理。

程序中如果没有发生异常,即 source.fail()为逻辑假,则将继续执行:

```
while(!source.eof())
{
  source.getline(line, sizeof(line));
  cout << line << endl;
}

source.close ( );
```

如果发生了异常,即 source.fail()为逻辑真,则抛掷的异常 throw argv[1] 将被

```
catch(char * s)
{
   cout <<"error opening the file " << s << endl;
   exit(1);
}
```

捕获,最后以执行"exit(1);"而告终。

可以将抛掷异常与处理异常放在不同的函数中。

例如,下面的程序定义一个除零异常:

```
//---------------------
//    ch21_2.cpp
//---------------------
#include <iostream>
using namespace std;
//---------------------
double Div(double, double);
//---------------------
int main(){
  try{
    cout <<"7.3/2.0 = "<< Div(7.3, 2.0)<< endl;
    cout <<"7.3/0.0 = "<< Div(7.3, 0.0)<< endl;
    cout <<"7.3/1.0 = "<< Div(7.3, 1.0)<< endl;
  }catch(double){
    cout <<"except of deviding zero.\n";
  }
  cout <<"That is ok. \n";
}//---------------------
double Div(double a, double b){
  if(b == 0.0)
    throw b;
```




```
    return a/b;
}//---------------------
```

运行结果为：

```
7.3/2.0 = 3.65
exception of dividing zero.
That is ok.
```

语句“cout <<"7.3/1.0=" << Div(7.3, 1.0) << endl;”没有被执行。

当调用函数表达式：

```
Div(7.3, 0.0)
```

时，控制转移到函数 Div()内执行，这时，“b==0.0”为逻辑真，被除 0 在数学上是无意义的，所以，程序中定义为异常。

由于发生了异常，函数 Div()被退栈处理，紧跟调用函数 Div()后面的语句：

```
cout <<"7.3/1.0 = " << Div(7.3, 1.0) << endl;
```

不再被执行。异常被

```
catch(double)
{
  cout << "except of deviding zero.\n";
}
```

所捕获，执行完异常处理，程序紧接着执行异常处理后面的语句：

```
cout <<"That is ok. \n";
```

如果程序中不发生异常，try 语句块中的第二条语句改为“Div(7.3,1.5);”，则运行结果为：

```
7.3/2.0 = 3.65
7.3/1.5 = 4.86667
7.3/1.0 = 7.3
That is ok.
```

程序在执行完 try 语句块之后，紧接着就执行 catch()语句块后面的语句。

21.4 异常的规则

以 catch 开始的语句块是异常处理程序，编写异常处理程序的规则是：

(1) 任意数量的 catch 语句块紧随出现在 try 语句块之后。在 try 语句块出现之前，不能出现这些 catch 语句块。例如：

```
int j;
double d;
char str[20];

class Coords
```

```
{
    public:
      Coords(double a, double b);
      //...
};

class String
{
   public:
      String(char * );
      //...
};

void f()
{
    try
    {
      //...
      throw 10;
      //...

      throw j;          //j 为 int 型
      //...
      throw d;         //d 为 double 型
      throw "abc";
      //...

      throw str;        //字符串
      //...

      throw Coords(1.0, 3.0);
      //...

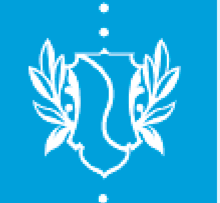
      throw String("def");
    }

    catch(int k)
    {
    //...
    }

    catch(double x)
    {
    //...
    }

    catch(char * ptr)
    {
    //...
    }

    catch(Coords c)
    {
    //...
```




```
    }

    catch(String s)
    {
    //...
    }

    cout <<"That is ok.\n";
}
```

在上例中 catch 语句块必须出现在 try 块之后，并且在 try 块之后可以出现多个 catch 语句块。

(2) 在 catch 行的圆括号中可包含数据类型声明，它与函数定义中参数声明起的作用相同。应把异常处理 catch 块看作是函数分程序。**跟在 catch 之后的圆括号中必须含有数据类型，捕获是利用数据类型匹配实现的**。在数据类型之后放参数名是可选的。参数名使得被捕获的对象在异常处理程序中被引用。

在上例中，抛掷异常：

“throw 10;” 和“throw j;” 被 catch(int j)的处理程序捕获；

“throw d;”被 catch(double x)捕获；

“throw "abc";”和“throw str;”被 catch(char * ptr)捕获；

“throw Coords(1.0,3.0);”被 catch(Coords c)捕获；

“throw String("def");”被 catch(String s)捕获。

捕获的原因是抛掷的数据类型与异常处理程序的数据类型相匹配。

我们在程序 ch21_2.cpp 中已经看到 catch(double)的声明中，参数名是省略的，在该异常处理中，没有用到参数名。

(3) 如果一个函数抛掷一个异常，但在通往异常处理函数的调用链中找不到与之匹配的 catch，则该程序通常以 abort()函数调用终止。

在上例中，在 try 块中，如果我们增加一个抛掷异常：

```
throw 'w';
```

则由于数据类型不匹配，而未能被任何 catch 块捕获，这时，系统用它自己的默认异常处理程序 abort()来做这项工作。

→ *在函数调用中，实参与形参可以通过相容类型的类型提升或自动转换来匹配：*

```
void g(int b)
{
  //...
}

void func()
{
  g('w');
  //...
}
```

g('w')将能顺利地匹配函数 g(int b)，但是抛掷异常与异常处理程序之间，是按数据类



型的严格匹配来捕获的。不允许类型转换,甚至下列代码:

```
try
{
  throw 20;
}

catch(unsigned int)
{
  //...
}
```

由于常量 20 的数据类型是 int,而不是 unsigned int,所以该抛掷异常无法被 catch(unsigned int)所捕获。

(4) 如果 catch 语句块执行完毕,则跟随最后 catch 语句块的代码(如果有的话)就被执行。

例如,上面例子中函数 f()的最后一条语句:

```
cout <<"That is ok.\n";
```

就是跟随最后 catch 语句块的代码。

21.5 多路捕获

多数程序可能有若干不同种类的运行错误,它们可以用异常处理机制。每种错误可与一个类、一个数据类型或一个值有关。这样,在程序中就会出现多路捕获。

例如,操作 String 类对象时,预设两个异常:

```
//---------------------
//  ch21_3.cpp
//---------------------
#include<iostream>
#include<cstring>
#include<exception>
using namespace std;
//---------------------
class String{
  char* p;
  int len;
  static int max;
public:
  String(char*, int);
  class Range{                    //在类中定义的异常类 1
  public:
    Range(int j):index(j){}
    int index;
  };
  class Size{};                   //异常类 2
  char& operator[](int k){
    if(k<0 || k>=len)
      throw Range(k);
    return p[k];
```

```
    }
}; //--------------------
int String::max = 20;              //静态成员初始化
//---------------------
String::String(char * str, int si){
  if(si < 0 || max < si)
    throw Size();

  p = new char[si];
  strncpy(p, str, si);
  len = si;
}//--------------------
void g(String& str){
  int num = 10;
  for(int n = 0; n < num; n++)
    cout << str[n];
  cout << endl;
}//--------------------
void f(){
  //代码区 1
  try{
    //代码区 2
    String s("abcdefghijklmnop", 10);
    g(s);
  }catch(String::Range r){
    cerr <<" -> out of range: "<< r.index << endl;
  //代码区 3
  }catch(String::Size){
    cerr <<"size illegal!\n";
  }catch(bad_alloc& e){ //new 分配失败异常处理
    cerr <<"bad allocation using new operator.\n";
    exit(1);
  }
  cout <<"The program will be continued here.\n\n";
  //代码区 4
}//--------------------
int main(){
  //代码区 5
  f();
  cout <<"These code is not effected by probably exception in f().\n";
}//--------------------
```

运行结果为:

```
abcdefghij
The program will be continued here

These code is not effected by probably exception in f().
```

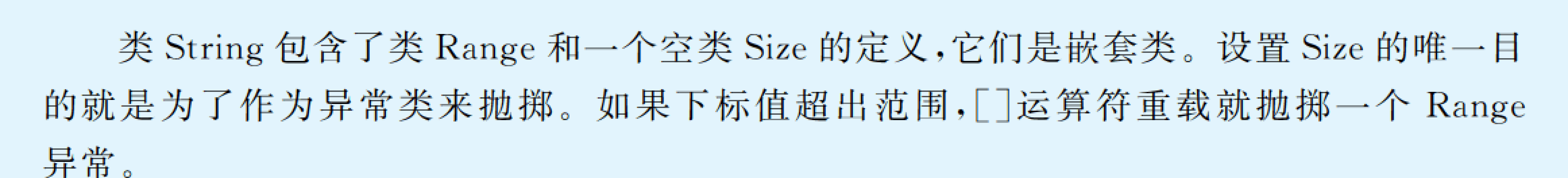
如果在代码区 2 中,构造 String 对象的定义为:

```
String s("abcdefghijklmnopqrstuvwxyz", 26);     //将抛掷 Size()异常
```

则运行结果为:

```
size illegal!
The program will be continued here

These code is not effected by probably exception in f().
```


类 String 包含了类 Range 和一个空类 Size 的定义，它们是嵌套类。设置 Size 的唯一目的就是为了作为异常类来抛掷。如果下标值超出范围，[]运算符重载就抛掷一个 Range 异常。

→ 在一个类定义的内部定义一个类，称为嵌套类。

嵌套类的成员函数和静态成员可以在包含该类的外部定义，但嵌套类的作用域在包含该类定义的内部。

如果 String 构造函数的参数 si 不在给定的数值范围内，它就抛掷一个 Size 异常。当函数 f()开始执行时，就首先执行标志为代码区 1 的代码。

如果代码区 1 抛掷一个异常，它只能被系统默认的异常处理程序 abort()所捕获，导致终止运行。因为它不在用户定义的异常区域。

在代码区 1 执行之后(没有发生异常)，在 try 块中代码区 2 就开始执行。如果这段代码直接或间接地抛掷一个异常，若该异常的数据类型与 catch 参数的数据类型匹配，跟在 try 块之后的 catch 块就可捕获这个异常。如果它未被捕获，该异常又只能被系统默认的异常处理程序 abort()捕获。

如果其中有一个 catch 块实际地捕获了所抛掷的异常，那么该 catch 块的代码就执行。如果这段代码没有执行终止运行的语句，没有返回语句，也没有再抛掷语句，那么就执行代码区 4 的代码。如果 catch 抛掷一个不跟实参的异常 throw，则由系统默认的异常处理程序来捕获。

异常处理完后，就执行代码区 4 的代码，以后，函数 f()正常结束。如果代码区 4 中有抛掷异常，则只能被系统默认的异常处理程序捕获。因为它也不在用户定义的异常区域内。

值得注意的是，C++ 自带标准异常类定义及默认异常处理。它在头文件 exception 中。其中当申请内存空间的 new 操作失败时，将抛掷 bad_alloc 异常，它是 except 异常类的子类。所以，ch21_3.cpp 中也定义了 bad_alloc 参数的 catch 语句块专门处理 new 操作失败的情况。

在主程序中，如果在执行代码区 5 的代码时，抛掷一个异常，则由系统默认的异常处理程序捕获。因为它不是用户定义的异常区域。

在调用 f()中，可能发生抛掷异常，被程序定义的异常处理程序所捕获。但是，从 f()返回后，其他部分的代码继续运行，不受影响。

→ catch(Size)并没有对 Size 对象指派名字。因为 Size 对象不含数据成员，也没有必要给捕获对象一个名字。

21.6 异常处理机制

在处理程序和语句之间的相互作用使异常在大型应用程序中变得复杂。通常人们希望抛掷被及时捕获，以避免程序突然终止。此外，跟踪抛掷很重要，因为捕获确定该程序的后继进展。例如，抛掷和捕获可以用来重新开始程序内的一个过程，或者从应用程序的一部分跳到另一部分，或者回到菜单。

例如，下面的代码说明了异常处理机制：




```
 void f ( )
 {
    try
    {
      g();
   }
    catch(Range)
    {
     //...
   }
   catch(Size)
   {
     //...
   }
   catch( ... )
   {
     //...
   }
}

void g()
{
   h();
}

void h()
{
   try
   {
     h1();
   }
   catch(Size)
   {
     //...
     throw 10;
   }
   catch(Matherr)
   {
     //...
   }
}

void h1()
{
     //...
     throw(Size);
   try
   {
     //...
     throw Range;

     h2();
     h3();
   }
   catch(Size)
   {
     //...
```

```
        throw;
      }
    }

    void h2()
    {
        //...
        throw Matherr;
    }

    void h3()
    {
        //...
        throw Size;
    }
```

处理程序的模式可由函数调用链中的异常处理来描述，见图 21-2。它显示包含 try 和异常处理程序的各个函数。它们使程序员能确定在一个应用程序中抛掷和捕获的模式。

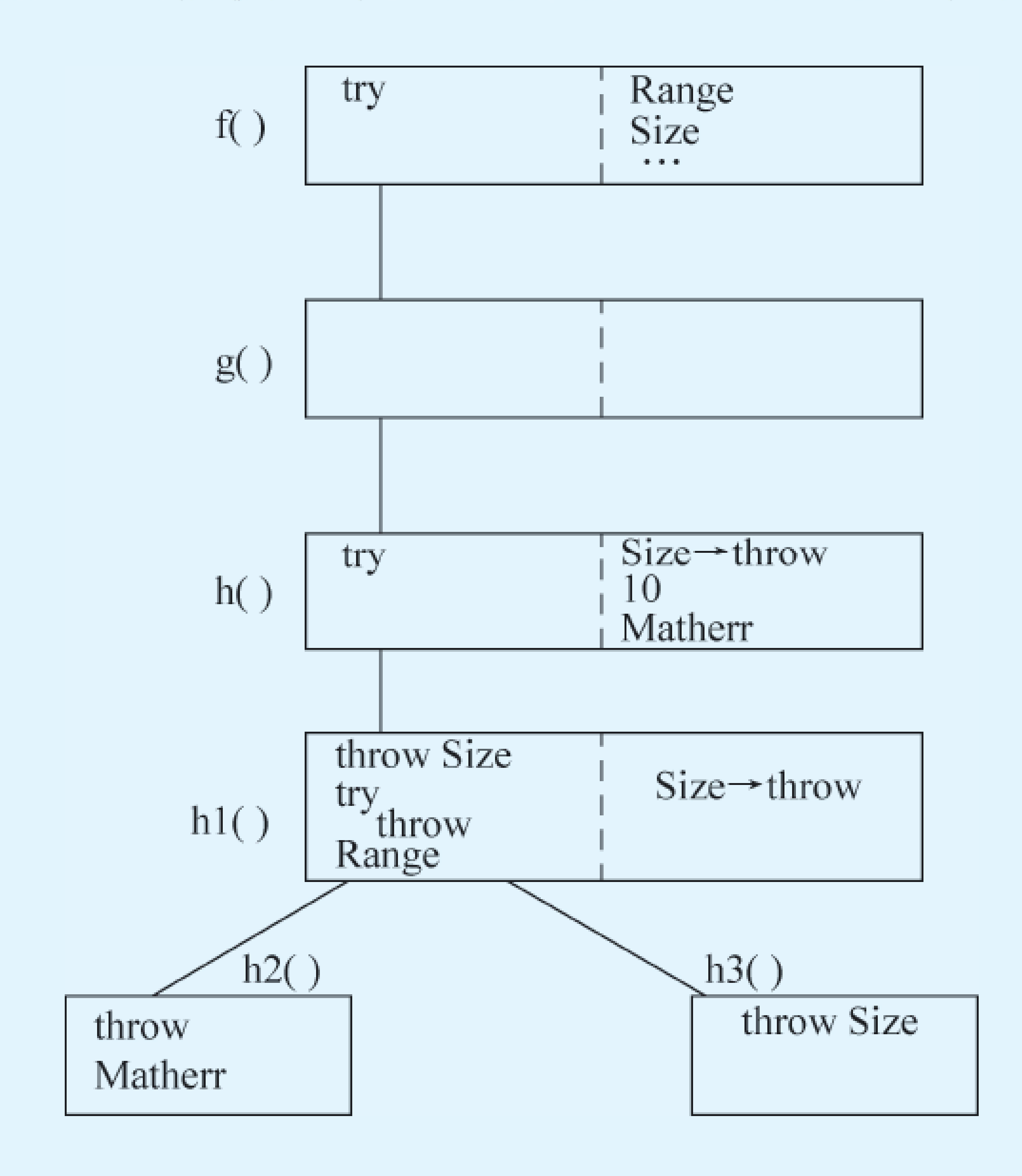


图 21-2　在函数调用链中的异常处理

在图 21-2 中每个函数以方框形式出现。每个方框分为两部分。左边部分表示该函数是否包含一个 try 语句定义，它也指出在 try 之前或在最后的异常处理程序之后的所有显式抛掷。在 try 块中的所有显式抛掷语句被表示成在 try 之下把 throw 缩进的形式。

方框的右边部分通过 catch 的数据类型列出各异常处理。图中表明一个异常处理中是否执行重新抛掷。重新抛掷时，处理程序后面是一个箭头和被抛掷对象的数据类型。

函数 f()中的 catch(...) 块，参数为省略号，定义一个“默认”的异常处理程序。通常这个处理程序应在所有异常处理块的最后，因为它与任何 throw 都匹配，目的是为避免定义的异常处理程序没能捕获抛掷的异常而使程序运行终止。

函数 h()中的 catch(Size) 块，包含有一个抛掷异常语句 throw 10，当实际执行这条语



句时，将沿着调用链向上传递，被函数 f()中的 catch(...) 所捕获。如果没有 f()中的 catch(...)，那么，异常将被系统的 terminate() 函数调用，后者按常规再调用 abort()。

函数 h1()中的抛掷 throw Size，由于不在本函数的 try 块中，所以只能沿函数调用链向上传递，结果被 h()中的 catch(Size)捕获。

函数 h1()中的抛掷 throw Range，在 try 块中，所以首先匹配 try 块后的异常处理程序，可是没有被捕获，因而它又沿函数调用链向上，在函数 f()中，catch(Range)块终于捕获了该抛掷。

函数 h1()中的 catch(Size)块，包含一个抛掷 throw，没有带参数类型，它表示将捕获到的异常对象重新抛掷出去，于是，它将沿函数调用链向上传递，在 h()中的 catch(Size) 块，捕获了该抛掷。

函数 h2()中的抛掷 throw Matherr，首先传递给 h1()中的 catch 块组，但未能被捕获，然后继续沿调用链向上，在 h()中的 catch(Matherr)块，捕获了该抛掷。

函数 h3()中的抛掷 throw Size，向上传递给 h1()中的 catch 块组，被 catch(Size)块捕获。

21.7 使用异常的方法

可以把多个异常组成族系。构成异常族系的一些示例有数学错误异常族系和文件处理错误异常族系。在 C++代码中把异常组在一起有两种方式：异常枚举族系和异常派生层次结构。

例如，下面的代码是一个异常枚举族系的例子：

```
enum FileErrors{nonExist, wrongFormat, diskSeekError, ...};

int f()
{
   try
   {
     //...
     throw wrongFormat;
   }

   catch(FileErrors fe)
   {
     switch(fe)
     {
       case nonExist:
         //...
       case wrongFormat:
         //...
       case diskSeekError:
         //...
     }
   }
   //...
}
```



在 try 块中有一个 throw，它抛掷一个 FileErrors 枚举中的常量。这个抛掷可被 catch(FileErrors) 块捕获到，接着后者执行一个 switch，对照情况列表，匹配捕获到的枚举常量值。

上面的异常族系也可按异常派生层次结构来实现，如下例所示：

```
class FileErrors{};

class NonExist :public FileErrors{};
class WrongFormat :public FileErrors{};
class DiskSeekError :public FileErrors{};

int f()
{
   try
   {
     //...
     throw WrongFormat;
   }
   catch(NonExist)
   {
     //...
   }
   catch(DiskSeekError)
   {
     //...
   }
   catch(FileErrors)
   {
     //...
   }
   //...
}
```

上面的各异常处理程序块定义了针对类 NonExist 和 DiskSeekError 的派生异常类对象，针对 FileErrors 的异常处理，既捕获 FileErrors 类对象，也捕获 WrongFormat 对象。

异常捕获的规则除了前面所说的，必须严格匹配数据类型外，对于类的派生，下列情况可以捕获异常：

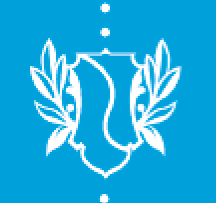
(1) 异常处理的数据类型是公有基类，抛掷异常的数据类型是其派生类；

(2) 异常处理的数据类型是指向公有基类的指针或引用，抛掷异常的数据类型是指向派生类的指针或引用。

→ 对于派生层次结构的异常处理，catch 块组中的顺序是重要的。因为“catch(基类)”总能够捕获“throw 派生类对象”，所以“catch(基类)”块总是放在“catch(派生类)”块的后面，以避免“catch(派生类)”永远不能捕获异常。

小结

为了检测异常，程序中使用 try、catch 和 throw 语句。异常处理使程序中错误的检测简单化，并提高程序处理错误的能力。



所谓异常是指程序中有运行错误。

程序应能检测错误:

try 语句使 C++能够进行异常检测;

catch 紧跟在 try 语句后面,可捕获异常;

throw 语句报告异常;

异常通过 throw 一个类型和 catch 的参数相匹配而捕获;

捕获异常后,程序将执行 catch 中的语句;

如果程序抛出一个不能捕获的异常,C++将执行默认异常处理函数。

练习

21.1 给 21.5 节中 ch21_3.cpp 添加一个“Pastm”异常类型的处理程序,如果[]运算符重载在 String 对象中检测到一个字符——按字典顺序在'm'之后的小写字母,该异常处理程序就在屏幕上显示一个错误。

21.2 设有下列类声明:

```
class A{
  public:
    A()
    {
      n = new int;
      init();
    }
  private:
    int n;
};
```

写出 init()引发异常的处理程序。

参 考 文 献

[1] Bjarne Stroustrup. C++程序设计语言(原书第 4 版)[M]. 北京：机械工业出版社，2017.

[2] Bjarne Stroustrup. C++程序设计：原理与实践(基础篇)(原书第 2 版)[M]. 北京：机械工业出版社，2017.

[3] Nicolai M. Josuttis. C++标准库[M].侯捷，译. 2 版. 北京：电子工业出版社，2015.

[4] Stephen C. Dewhurst. C++覆辙录[M].高博，译. 北京：人民邮电出版社，2016.

[5] Bjarne Stroustrup. C++语言的设计与演化[M]. 裘宗燕，译. 北京：机械工业出版社，2002

[6] Bruce Eckel. C++编程思想(两卷合订本)[M].刘宗田，译. 北京：机械工业出版社，2011.

[7] Andrew Koenig，Barbara Moo. C++沉思录[M].黄晓春，译. 北京：人民邮电出版社，2010.

[8] Stephen Prada. C++Primer Plus 中文版[M].6 版.张海龙，等译. 北京：人民邮电出版社，2012.

[9] 宛延闿. C++语言和面向对象程序设计[M].2 版.北京：清华大学出版社，1998.

[10] Stephen R. Davis. C++编程指南(续篇)[M]. 卢凌云，等译. 北京：电子工业出版社，1997.

[11] Bruce Eckel. C++深入浅出[M]. 侯雪萍，等译. 北京：学苑出版社，1994.

[12] Scott Meyers. C++编程技巧[M]. 陈迅雷，等译. 上海：上海科学普及出版社，1994.

[13] Stephen Blaha. 最新 C++应用编程技巧[M]. 孟庆昌，等译. 北京：国防工业出版社，1997.

[14] Krisjamsa. 新编 C++自学教程[M]. 凌涛，等译.北京：电子工业出版社，1996.

[15] Namir C. 面向对象的编程指南[M]. 宋炎，等译.北京：电子工业出版社，1997.

[16] Jesse Liberty. C++自学通[M]. 路明，等译.北京：机械工业出版社，1997.

[17] Herbert Schildt. C++从入门到精通[M]. 马力文，等译. 北京：学苑出版社，1994.

[18] William Ford，William Topp. 数据结构[C++语言描述][M]. 北京：清华大学出版社，1997.

[19] Jesse Liberty. C++程序设计轻松入门[M]. 张文旭，等译. 北京：机械工业出版社，1996.

[20] Microsoft. 程序设计范例与教学参考[M]. 袁勤勇，译. 北京：学苑出版社，1994.

[21] Deitel H M，Deitel P J. C/C++程序设计大全[M]. 薛万鹏，等译. 北京：机械工业出版社，1997.

[22] 沈纪新. Visual C++使用速成[M]. 北京：清华大学出版社，1997.

[23] Michael Hyman，Bob Arnson. 学用 Visual C++ 5[M]. 马岚，等译. 北京：电子工业出版社，1998.

[24] 谭浩强. C 语言程序设计教程[M]. 北京：高等教育出版社，1992.

[25] Kris Jamsa. C/C++使用技巧 1001 例[M]. 魏津，等译. 北京：电子工业出版社，1996.

[26] Herbert Schildt. 最新 C++语言精华[M].杨长虹，等译. 2 版. 北京：电子工业出版社，1997.

[27] Steve Oualline. 使用 C++编程大全[M]. 辛运帏，等译. 北京：电子工业出版社，1997.

[28] Peter Vander Linden. C 程序设计奥秘[M]. 张自力，译. 昆明：云南科技出版社，1998.

[29] Herbert Schildt. C++自学教程[M]. 石桥林，等译. 北京：学苑出版社，1994.

[30] 张国峰，等. C++程序设计实用教程[M]. 北京：清华大学出版社，1996.

[31] Cameron Hughes. C++iostream 面向对象 I/O 程序设计[M]. 尤晓东，等译. 北京：电子工业出版社，1997.

[32] Kaare Christian. Microsoft C++程序设计指南[M]. 北京：清华大学出版社，1994.

[33] Greg Perry. C++程序设计教程[M]. 北京：清华大学出版社，1994.

[34] 王燕. 面向对象的理论与 C++实践[M]. 北京：清华大学出版社，1997.

[35] Tom Swan. C++编程秘诀[M]. 宋建云，译. 北京：电子工业出版社，1994.